(4)

VLSI Memo No. 88-475
September 1988

AN INTELLIGENT PROCESS FLOW LANGUAGE EDITOR

Rajeev Jayavant

Abstract

A process flow language allows a user to specify the process used to fabricate integrated circuits on a silicon wafer. By using a flow language, a designer can modify his process more easily and processing equipment can be reconfigured for use with different processes.

While the benefits gained from using a process flow language have been discussed frequently, one major drawback of using a flow language has been overlooked: users must code their process flows in the flow language. This may not seem like a disadvantage, but it is rather difficult to convince people to so something they have never done before. Furthermore, the process must be in a language that faintly resembles Lisp, not a very appealing thought for users whose primary interest is in processing wafers, not programming computers. Thus there is a severe need for some tool to facilitate the coding of processes flows in the process flow language.

Various types of programming aids have been used in the past to facilitate software development: syntax checkers, semantic checkers, preprocessors, and intelligent editors. The process flow editor combines attributes from all of these. The primary difference between the process flow editor and conventional editors is that the flow editor presents the flow to the user in a format that is very different from the format seen by applications accessing the flow. (KR) ←

By using a different format in presenting the flow to the user, most people will not have to learn the Lisp-like syntax of the flow language and can concentrate on what they really want to do -- specify a process flow. The current implementation of the flow editor uses a forms-based interface to present the flow as a collection of nested operations. A forms-based interface is appealing because it facilitates the design of the editor while providing an interface that lab users will recognize from several CAFE applications. The use of forms also allows the flow editor to highlight the decomposition of the process flow into parameterized operations, thereby providing a more informative view of the flow to the user.

Syntax and semantic checking is performed as the user enters the flow. The time required to code a process flow is reduced since many common errors are caught as they are made rather than being discovered at a later time by an interpreter.

Microsystems Research Center
Room 39-321

Massachusetts Institute of Technology

Cambridge
Massachusetts
02139

Telephone
(617) 253-8138

An Intelligent Process Flow Language Editor

by

Rajeev Jayavant

S.B. Electrical Engineering, Massachusetts Institute of Technology
(1986)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Science

and

Electrical Engineer

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1988

Signature of Author .....
Department of Electrical Engineering and Computer Science
August 31, 1988

Certified by .....
Donald E. Troxel
Professor of Electrical Engineering
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

An Intelligent Process Flow Language Editor

by

Rajeev Jayavant

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 1988, in partial fulfillment of the
requirements for the degrees of
Master of Science
and
Electrical Engineer

Abstract

A process flow language allows a user to specify the process used to fabricate integrated circuits on a silicon wafer. By using a flow language, a designer can modify his process more easily and processing equipment can be reconfigured for use with different processes.

While the benefits gained from using a process flow language have been discussed frequently, one major drawback of using a flow language has been overlooked: users must code their process flows in the flow language. This may not seem like a disadvantage, but it is rather difficult to convince people to do something they have never done before. Furthermore, the process must be in a language that faintly resembles Lisp, not a very appealing thought for users whose primary interest is in processing wafers, not programming computers. Thus there is a severe need for some tool to facilitate the coding of processes flows in the process flow language.

Various types of programming aids have been used in the past to facilitate software development: syntax checkers, semantic checkers, preprocessors, and intelligent editors. The process flow editor combines attributes from all of these. The primary difference between the process flow editor and conventional editors is that the flow editor presents the flow to the user in a format that is very different from the format seen by applications accessing the flow.

By using a different format in presenting the flow to the user, most people will not have to learn the Lisp-like syntax of the flow language and can concentrate on what they really want to do – specify a process flow. The current implementation of the flow editor uses a forms-based interface to present the flow as a collection of nested operations. A forms-based interface is appealing because it facilitates the design of the editor while providing an interface that labusers will recognize from several CAFE applications. The use of forms also allows the flow editor to highlight the decomposition of the process flow into parameterized operations, thereby

providing a more informative view of the flow to the user.

Syntax and semantic checking is performed as the user enters the flow. The time required to code a process flow is reduced since many common errors are caught as they are made rather than being discovered at a later time by an interpreter.

KEYWORDS: Process Flow, Intelligent Editor, Wafer Fabrication

Thesis Supervisor: Donald E. Troxel
Title: Professor of Electrical Engineering

Acknowledgments

I would like to thank Professor Troxel for his invaluable insights and infinite patience. Somehow he always knew which path to follow, and I doubt this project would have been completed for several more months without his guidance.

I would also like to thank the members of the CAF project for their help during the past two years. Special thanks to Mike McIlrath and Mike Heytens for putting up with my constant questions about how things should be done or complaints about things not working the way I would like them to. I am also very grateful to Duane Boning for providing a coding of the CMOS Baseline flow in the process flow language – it proved to be the sole hardcopy reference for the definition of the process flow language.

Finally I would like to thank all my friends that have put up with me during some trying times. Very special thanks to Saed Younis who always managed to convince me that things weren't all that bad when they seemed darkest.

This research was supported in full by the M.I.T. Computer Aided Fabrication (CAF) project at the facilities of the Cognitive Information Processing Group (CIPG) of the Research Laboratory for Electronics.

dedicated to my family

Contents

1 Introduction 11

2 Previously Developed Programming Aids 14

2.1 Pretty Printers 14

2.2 Semantic Checking 15

2.3 Preprocessors 15

2.4 Interactive Tools 16

2.4.1 The Cornell Program Synthesizer 17

2.4.2 GNOME 17

2.4.3 PECAN 19

3 The Process Flow Language 20

3.1 Operations 21

3.2 Sequences 22

3.3 Definitions 24

4 The User Interface 25

4.1 Design Considerations 25

4.1.1 Alternate Syntax 26

4.1.2 Template-based vs. Free-form Editors 27

4.2 Implementation 28

4.2.1 User's Perspective 29

4.2.2 Implementor's Perspective 30

5 Semantic Checking **38**

5.1 Parameter Values in Operation References 39

5.1.1 Number of Parameter Values 39

5.1.2 Validity of Parameter Values 40

5.2 Type Checking Within Operations 43

5.3 Type Checking Within Sequences 43

5.4 Type Inference Mechanisms 44

5.4.1 Design Methodologies 44

6 The Need for Flexibility **47**

6.1 Overall Design 47

6.2 Multiple Slots 49

6.3 Adding Functionality 51

7 Results and Directions for Future Work **53**

7.1 Implementation Status 53

7.2 Enhanced Semantic Checking 54

7.3 Hiding Lisp Syntax 56

7.4 Support for the X Window System 57

7.5 Database Support 58

A The *PFLE* User Manual **59**

A.1 Introduction 60

A.2 Entering and Exiting *PFLE* 61

A.3 Basic Commands 62

A.4 The Top-Level View 63

A.5 Editing Definitions 64

A.6 Editing Operations 65

A.6.1 Parameterized Operations 66
A.6.2 Descriptive Comment 66
A.6.3 References to Other Operations 66
A.7 Creating a Simple Process Flow 69

B Users' Guide to *Fabform* **74**
B.1 Introduction 76
B.2 Basic Operation 76
B.2.1 Screen Layout 76
B.2.2 Command Summary 77
B.3 Cursor Movement 79
B.4 Editing Field Entries 80
B.5 Validation of Field Entries 80
B.5.1 Unrestricted Fields 80
B.5.2 Integer Fields 80
B.5.3 Floating Point Fields 80
B.5.4 Lisp Expression Fields 81
B.5.5 Oneof Fields 81
B.5.6 Default Values 81
B.5.7 Date Fields 81
B.5.8 Read-only Fields 81
B.6 Saving Entered Field Values 82
B.7 Exiting *Fabform* 82
B.7.1 Temporary Suspension 82
B.7.2 Permanent Exit 82

C Programmers' Guide to *Fabform* **84**
C.1 Introduction 86
C.2 Interaction with *Fabform* 86

C.3 Template File Format 88
C.3.1 Positioning Commands 88
C.3.2 Field Definition Commands 88
C.3.3 Operation Block Delimiters 90
C.4 Parameter File Format 90
C.5 Invoking *Fabform* 94

D The Procedural Interface to *Fabform* **97**
D.1 Introduction 99
D.2 The *Fabform* Interface 100
D.2.1 Function Definitions 100
D.2.2 *Fabform* Options 103
D.2.3 Return Value 103
D.3 The Function Interface 104
D.3.1 The Lisp Function Interface 106
D.4 Signal Handling 106
D.5 Utility Routines 107
D.6 Linking the *Fabform* Subroutine 113

List of Figures

3-1 Coding of the *photomask* operation using the process flow language . 22

3-2 Coding of the *resist-develop* operation using the process flow language 23

3-3 Coding of the *furnace-rampup-treatment* using the process flow language 23

3-4 Coding of the *gateoxtube* definition using the process flow language . 24

4-1 *PFLE* representation of *photomask* 31

4-2 *PFLE* representation of *resist-develop* 32

4-3 *PFLE* representation of *furnace-rampup-treatment* 33

4-4 *PFLE* representation of *gateoxtube* 34

5-1 Choices available upon discovery of an undefined identifier 41

5-2 Menu of options for *:treatment* slot containing *:furnace* 46

6-1 Overall Design of *PFLE* 48

A-1 Menu for selecting type of object to create 70

A-2 *simple* with fields for two references 71

A-3 *simple* with reference to *grow-field-oxide* 71

A-4 *pattern-active-area* with parameter defined 72

A-5 Completed definition of *simple* 72

Chapter 1

Introduction

The sequence of operations, or treatments, performed on a silicon wafer in order to fabricate integrated circuits is often referred to as a process flow. The development of a language for describing a process flow is an integral part of the CAF[1] effort at M.I.T. Without the aid of a flow language, a typical process flow may be described in a document several hundred pages in length that can only be understood by the process engineer that created it. Using a process flow language to specify the steps necessary to fabricate an integrated circuit on a silicon wafer provides several benefits:

- The process flow language provides a standard format for defining process flows. Any process engineer (or an interpreter that operates on the flow language) should be able to understand a flow coded in the process flow language. Thus the flow language serves as a documentation tool, allowing people other than the creator of the flow to fully understand the fabrication procedure.

- The same representation can be used for simulation and fabrication. Different interpreters can use the flow language description to perform different tasks. Once a process flow has been successfully simulated, it can be immediately

[1] Computer Aided Fabrication

tested on the fabrication line with minimal effort.

- Processing equipment can be more easily reconfigured to run several processes. Since the process flow is encoded in machine-readable form, it becomes possible to build systems that download recipes to equipment based on what will be processed next. A single fabrication facility can support multiple processes simultaneously.

- By precisely documenting the steps in the process flow and recording measurements made during fabrication, repeatability of fabrication results is improved. Tracing sources of problems leading to poor yields is also facilitated by the availability of information.

- The process flow description can be used to schedule the use of equipment in the lab. The scheduler would be yet another interpreter using the common flow language description.

The one major drawback of using a flow language is that users must code their process flows in the flow language. This may not seem like a *disadvantage*, but it is rather difficult to convince people to do something they have never done before. Furthermore, the process must be in a language that faintly resembles Lisp, not a very appealing thought for users whose primary interest is in processing wafers, not programming computers. Thus there is a severe need for some tool to facilitate the coding of process flows in the process flow language.

This thesis describes one such tool: an intelligent process flow language editor. We begin by first examining some of the tools that have been previously developed to simplify software development. The process flow language editor, or *PFLE*, combines features from many of the tools described in chapter two, however, it differs from most previously developed expert editors in one major respect: the process flow is presented to the user in a format that is very different from the syntax of the process flow language. By using a different format, we can hide much

of the Lisp-like syntax of the flow language from users who may be intimidated by the sight of parentheses. Chapters three and four discuss the syntax of the process flow language and the format presented to the user.

Chapter five discusses the semantic checking abilities of the editor and the design methodologies supported by *PFLE*. Chapter six considers the issue of flexibility in the design of the editor and explores some of the implementation issues related to that requirement of flexibility. *PFLE* is designed provide enough flexibility to form a platform on which to build more advanced *expert systems* for developing process flows.

Finally, chapter seven examines the present state of the flow language editor and considers possible areas of improvement in future implementations. One area of concern is how well the flow editor caters to different types of users. *PFLE* is intended to be used by both novices and experienced programmers. It should provide enough hand-holding to guide novices yet these hand-holding mechanisms should not hamper experienced programmers.

The appendices provide additional information which may be useful to someone that is planning to use the flow language editor or is considering building a simila piece of software. Appendix A is the user manual for the process flow language editor. Appendices B thrc ugh D are the manuals for *Fabform*, the generalized forms-based user interface upon which the flow language editor is built.

Chapter 2

Previously Developed Programming Aids

We begin the task of designing a tool for simplifying the coding of process flows in the flow language by first examining previous efforts at simplifying coding of programs. The process flow language contains many attributes found in familiar programming languages, thus a study of existing programming aids should provide some useful insights on how to begin designing *PFLE*. This basic design can then be extended to cater to the special needs of the process flow language.

2.1 Pretty Printers

Various types of programming aids have been used in the past to facilitate software development. The simplest such tools are the *pretty-printers* which reorganize the source code to improve readability. Hopefully, the reorganization will highlight the framework of the program and reveal syntax-related inconsistencies between the code and what the programmer intended. Thus the programmer can spend more time thinking about what he is trying to accomplish rather than waste his time figuring out what his code really means. The major disadvantage of pretty-printers

is that they are non-interactive. The utility is fed a source file and produces an output file containing the reorganized source code. Nevertheless, pretty-printers have proven to be useful programming aids and are available for a number of languages (e.g., the Unix **indent** utility for C and the CLU indenter included with the CLU development system).

2.2 Semantic Checking

Unfortunately syntax-checking alone will not detect a large number of coding errors. A syntax checker, for example, does not know anything about the type of various identifiers in a program, or whether an identifier is valid within a particular section of code. A semantic checker is needed to determine whether it is legal to perform operations on specific identifiers. It can also be argued whether the semantic checker or the syntax checker should be responsible for ensuring that the proper number of arguments is passed to a procedure (though the semantic checker certainly has the responsibility of determining whether the types of the arguments are correct). An example of a semantic checker is **lint**, the Unix utility for validating C programs.

The trend in the evolution of programming languages and compilers has been to detect as many errors at compile time as opposed to run time. While the Unix C compiler does some type-checking, it does not, in general, check the number or types of arguments passed to procedures. A compiler for a more strongly-typed language, such as CLU, does perform the rigorous checking of arguments. A C programmer must use **lint** to obtain rigorous checking. Thus as languages and compilers evolve, the semantic checkers have moved from stand-alone utilities into the compilers.

2.3 Preprocessors

A completely different class of programming aids can be grouped under the category of preprocessors. A preprocessor is a tool which transforms an input file from

one format into another. In some sense, a pretty-printer can be thought of as a preprocessor, however a preprocessor generally allows the input to be specified using a slightly different syntax from the output. For example, a macro preprocessor allows the input file to contain macro definitions and instances in which the macros are used. The output file will not contain the macros but rather the result of expanding the macros. Thus the output file can be fed to some other tool that doesn't understand macros, allowing the programmer to use a more suitable syntax (e.g., macros). Another example of a preprocessor is ratfor, the so-called *rational* FORTRAN preprocessor which allows the use of structured constructs such as while loops to be used in a FORTRAN program.

2.4 Interactive Tools

The next step in the *evolution of programming* aids has been the development of increasingly intelligent editors. Emacs provides a good example of such an editor. In addition to providing interactive indentation mechanisms as mentioned above, it provides a certain amount of syntax checking[1]. In Lisp mode, it indicates pairs of matching parentheses as the user types close parentheses. In C mode, it goes one step further by matching pairs of braces and brackets as well.

Language-sensitive editors (e.g., GNOME, POE) provide more rigorous checking of syntax. In addition, they facilitate programming for novices by allowing the programmer to insert code fragments in terms of fundamental syntactic blocks rather than forcing him to build the blocks from their individual components. [2] The following subsections examine a few language-sensitive editors in greater detail.

The advantage of making editors increasingly language-sensitive is to discover as many errors as early as possible (program creation time as opposed to run-time

[1] Perhaps syntax highlighting is a more appropriate description

[2] In POE (A Pascal-Oriented Editor [Fischer84]), for example, the user simply needs to enter "IF" to force the editor to insert a complete "IF-THEN-ELSE" block.

or even compile-time). Just as increased semantic checking at compile-time versus run-time reduced the time required to develop software, increased semantic at code creation time will reduce coding time even further.

2.4.1 The Cornell Program Synthesizer

The Cornell Program Synthesizer is one of the first syntax-directed editing environments to see any degree of widespread use. The syntax-directed editor is, in fact, just one component of an integrated programming environment for creating, editing, executing, and debugging programs written in PL/CS (a dialect of PL/I). We will, however, concentrate on the editing features of the Program Synthesizer.

The editing environment is syntax-directed in which a program is viewed (conceptually) in terms of a syntax tree rather than as text. A program is created or modified by inserting or deleting predefined templates at appropriate positions in the program. Virtually all editing is done in terms of templates – only numbers, identifier names, and character strings are entered character by character. The grammar is encoded into the templates, thus the editor only allows templates to be inserted or deleted in a manner that results in a syntactically correct program. The original implementation of the Program Synthesizer [Teitelbaum81] did not perform any semantic checking. The Synthesizer Generator [Teitelbaum84] creates a Program Synthesizer-like editor for an arbitrary language. The language must be specified in terms of an attribute grammar, allowing some semantic checking information to be specified. The Berkeley process flow language editor, SEPS [Sedayao88], is created using the Synthesizer Generator, though it does not perform any semantic checking in its current implementation.

2.4.2 GNOME

GNOME [Garlan84] provides a syntax-directed programming environment targeted primarily at novice programmers. The family of four editors (each catering to a

different language) is used by students in the introductory programming class at Carnegie Mellon University, providing ample feedback on the utility of the programming environment provided by GNOME. The individual editors within GNOME are are based on ALOE (also developed at CMU) which generates an editor from a grammar for the language to be edited.

Like the Cornell Program Synthesizer, GNOME is a complete programming environment, allowing programs to be run from within the editor. There are, however, a number of important differences between GNOME and the Program Synthesizer which are required if a syntax-directed editor is to prove useful to a novice programmer.

Although both editors view a program in terms of a syntax tree, the Synthesizer reflects this view in the way a user traverses a program. The cursor can only be moved from one template to the next. To move to a template within another template (e.g. the predicate within an if-then-else expression) requires a separate command. GNOME, on the other hand, exercises less rigorous control on cursor movement, allowing the cursor to be moved onto any editable location within a program, much like a text editor. Novice programmers tend to think of programs as text and can be confused by a structure-oriented editor like the Synthesizer.

Some of the other features of GNOME cater particularly well to novice programmers in general and to a teaching environment in particular. GNOME attempts to enforce programming methodologies by making it easier to code programs the "right" way. For example, GNOME simplifies the task of declaring all identifiers before defining the procedure body. The other notable feature is that GNOME not only performs semantic checking, it attempts to help the user correct semantic errors, if possible. For example, if GNOME detects the use of an undeclared identifier, it asks the user whether it should define the identifier.

2.4.3 PECAN

PECAN [Reiss84] also provides an integrated programming environment, however, it is unique in its user interface. Although the program is stored internally as an abstract syntax tree, PECAN provides multiple views of the program. For example, some users will prefer the standard textual description which may be edited using a structural editor (similar to the one provided by the Cornell Synthesizer). Others may prefer a graphical representation in the form of a flow chart or a module interconnection diagram. Thus PECAN decouples the representation a program is stored in from the representation seen by the user.

Chapter 3

The Process Flow Language

The Process Flow Language is embedded in Common Lisp, thus it inherits many of the qualities of Lisp. The most noticeable of these qualities are the use of parenthesized expressions and the use of keywords. The flow language also uses the define construct found in Scheme. Building an editor capable of performing semantic checking of arbitrary Lisp expressions would be a formidable task since types of objects cannot be easily determined until runtime.

The process flow language editor simplifies its task by concentrating its efforts on a subset of the flow language. By choosing the proper subset, the editor retains enough power to specify virtually any process flow while gaining the ability to provide very thorough syntax and semantic checking mechanisms. In some cases, expressing a flow using only the subset of the flow language may lead to a flow description that is more cumbersome than one coded using the full power of the flow language. A cumbersome description, however, is infinitely preferable to no description at all (which is what would be available without the aid of an editor like *PFLE*). Thus the flow language editor views a process flow in terms of three types of objects: operations, sequences, and definitions.

3.1 Operations

An operation is the fundamental unit of a process flow. The entire process flow is merely a special type of operation that is called a **flow**. Figures 3-1 and 3-2 show examples of two operations from an encoding of the CMOS baseline process flow. An operation consists of:

- An optional comment. The comment is used to describe what function the operation performs. *PFLE* displays this comment whenever the operation is referenced in another operation or in a sequence.
- Zero or more other operations or sequences of operations. Operations may be nested to arbitrary depths. The *photomask* operation shown in figure 3-1 references the operations *hmds-prime*, *resist-coat*, *resist-expose*, *resist-develop*, *resist-bake*, and *resist-inspect*.
- Multiple slots. The *photomask* operation contains slots for **:settings** whereas the *resist-develop* operation in figure 3-2 contains slots for **:change-wafer-state** and **:settings**.

The process flow language does not specify how many slots an operation must have or what those slots should be. The semantics associated with any particular slot are determined entirely by the interpreter that operates on the process flow. An interpreter should ignore any slots that it does not understand, and it should not require a slot to be specified at all times. The interpreters based on the two-stage process model, for example, work with three types of slots:

:change-wafer-state description of the change in wafer state as a result of performing the operation

:treatment the wafer treatments that will lead to the desired change in wafer state

:settings the machine settings required to produce the desired treatments (and therefore to produce the desired change in wafer state)

A particular slot may contain primitives or sequences of primitives appropriate to that slot. The primitives for a given slot are determined solely by the interpreters that access that slot. The actual syntax used within a slot is also determined by the interpreters, thus the syntax can vary from slot to slot.

Operations can be parameterized by using the **define** construct. Default values can also be specified for parameters as illustrated in the photomask operation in Figure 3-1. Arguments to a parameterized operation are passed by keyword, thereby eliminating the need to remember the proper order of parameters.

```
(define (photomask mask (resist :positive-resist)
                        (dswjob :unknown))
   (operation
       "Standard photomask step"
       hmds-prime
       (resist-coat :resist resist)
       (resist-expose :mask mask :dswjob dswjob)
       resist-develop
       resist-bake
       resist-inspect
       (:settings
            (:time-required (:hours 3 :minutes 35)))))
```

Figure 3-1: Coding of the *photomask* operation using the process flow language

3.2 Sequences

Sequences are lists of primitives or sequences of primitives. All of the primitives within a sequence must be of the same type, i.e. all of the primitives must be

```
(define resist-develop
   (operation
      "The exposed wafers are loaded on to the GCA developing
track.  Resist is developed using a spray-develop recipe and
subsequently hard baked at 200 C for 1 min.  KTI 934 Alkali ion free
developer (premixed 1-1) is used."
      (:change-wafer-state
         (:develop))
      (:settings
         (:recipe 21)
         (:time-required (:hours 0 :minutes 50)))))
```

Figure 3-2: Coding of the *resist-develop* operation using the process flow language

appropriate for a single type of slot in an operation. Figure 3-3 shows the *furnace-rampup-treatment* which is a sequence of :treatment primitives. Thus *furnace-rampup-treatment* may be used within a :treatment slot of an operation or within another sequence of :treatment primitives. Sequences consisting of operations are also possible, though they are rarely used since they are essentially equivalent to operations without any slots.

```
(define (furnace-rampup-treatment temperature)
   (sequence
     (:furnace :temperature 800 :time (:minutes 10)
               :ambient :n2)
     (:furnace :temperature 800 :ambient :n2 :time
               (:minutes (/ (- temperature 800) 10))
               :temp-rate 10)
     (:furnace :temperature temperature
               :time (:minutes 10) :ambient :n2)))
```

Figure 3-3: Coding of the *furnace-rampup-treatment* using the process flow language

Like operations, sequences may have optional comments (*furnace-rampup-treatment*

does not have a comment) and can be parameterized using the define construct.

3.3 Definitions

From a Lisp or Scheme programmer's point of view, all of the examples are considered to be definitions. The process flow editor, however, regards definitions as a construct for defining *constants*. Figure 3-4 shows the definition of *gateoxtube*, the furnace tube that is used for growing gate oxide on a silicon wafer. Experienced programmers can also use definitions to reference more advanced features of the process flow language.

```
(define gateoxtube "tubeA1")
```

Figure 3-4: Coding of the *gateoxtube* definition using the process flow language

Chapter 4

The User Interface

The process flow editor combines attributes from all of the programming aids described in chapter 2: syntax checkers, semantic checkers, preprocessors, and intelligent editors. A notable difference between the process flow editor and conventional editors is that the flow editor presents the flow to the user in a format that is very different from the format in which it is actually stored [1]. Thus the process flow editor is like a preprocessor in that it converts the user's input into the syntax of the flow language. It differs from a preprocessor, however, in that the process flow is only stored using the flow language syntax and never in the *preprocessed* form. The editor is capable of converting from one form to the other when flows are loaded and saved. Since the flow is always saved in the syntax of the flow language, the flow can be edited using other editors, if desired.

4.1 Design Considerations

The design of the user interface of *PFLE* can be decomposed into two subproblems:

1. choosing the syntax to present to the user

[1] or the format that is used by programs accessing the flow

2. designing the method of interacting with the user (e.g. free-form vs. template-based entry)

Since most editors present the syntax of the language directly to the user, the design of the user interface is reduced to the second subproblem. In the case of *PFLE*, however, the choice of syntax presented to the user is perhaps even more important than the method of interaction with the user.

4.1.1 Alternate Syntax

By using a different format in presenting the flow to the user, most people will not have to learn the Lisp-like syntax of the flow language and can concentrate on what they really want to do – specify a process flow. Hiding the Lisp-like syntax is particularly important when dealing with users who are somewhat computer-phobic to begin with and refuse to have anything to do with programming.

The major issue in choosing the syntax to present to the user is to find a format that allows the user to specify the important concepts of the flow in a simple manner. The previous chapter describes the first step towards finding this syntax – choosing a subset of the process flow language that is as simple as possible while retaining enough expressive power to code most process flows. Thus the view of the flow presented to the user is based on the concept of parameterized operations.

A process flow consists of a number of operations, each of which may take zero or more parameters. Each operation, in turn, may be composed of other operations. This hierarchy of nested operations eventually bottoms out in a set of primitive operations defined by the various interpreters that will process the flow. In this view, **sequences** are special operations that do not have multiple slots but can be used within a slot of an operation. The semantic checker (described in the next chapter) is responsible for ensuring that **sequences** and **operations** are used in the proper context. The user only has to worry about two types of objects: operations and definitions.

PFLE provides a simpler, higher-level view of the process flow language. Instead of learning the intricacies of Lisp, the process engineer only needs to learn simpler the concepts of parameterized operations and definitions. In rare cases, though, the view presented by *PFLE* may lack the expressive power required to represent a segment of the process flow. In such cases, it will be necessary to code the segment directly in the process flow language (either by using a definition or by using a different editor). The situation is analogous to one faced by a Lisp programmer who finds that he must code some segments of his program in C because Lisp lacks the functionality for performing some low-level operations. Experience with the CMOS baseline process flow has shown that the subset of the flow language supported by *PFLE* is more than adequate to express the entire flow. Since most users will create flows by choosing from predefined operations (e.g. those defined within the CMOS baseline flow), *PFLE* will prove to be a very effective tool.

Another consideration in choosing the syntax to present to the user is the amount of effort required to learn the syntax. Since one of the main goals of *PFLE* is to allow novice users to create process flows with minimal effort, the syntax seen by the user should be similar to something he is already familiar with (and not frightened by). Forcing the user to learn a completely alien syntax may be even less productive than forcing him to learn Lisp, though he may be more willing to learn a syntax that isn't littered with parentheses.

4.1.2 Template-based vs. Free-form Editors

A template-based editor is one in which text is inserted or deleted in blocks rather than character by character. The blocks normally correspond to syntactic constructs. For example, a template-based editor for C or Pascal may insert or delete entire if-then-else expressions via a single keystroke. Template-based editors also tend to restrict where the cursor may be positioned and where templates may be inserted.

Free-form editors, on the other hand, resemble generic text editors. The text is inserted or deleted character by character. Cursor movement is normally not restricted, though an editor with syntax and semantic checking may restrict insertion or deletion of text in certain places.

Both types of editors can provide the basis for building an expert editor. Free-form editors have the advantage that they resemble the text editors that users are familiar with. Making minor changes is easy since a *program* can be modified on a character-by-character basis. Making minor changes using a template-based editor may require considerably more work since all changes must be made by inserting and deleting blocks of text. A minor modification could involve the deletion and reinsertion of text that wasn't affected by the change. Entering code, however, tends to be much faster using a template-based editor. Since text is inserted in blocks, large amounts of text can be entered using a few keystrokes.

From an implementation standpoint, building a syntax and semantic checking editor based on a template-based editor is much easier than building one based on a free-form editor. Since a template-based editor inserts or deletes text only in blocks, minimal effort is required to ensure that blocks are inserted and deleted in such a way that the resulting program remains syntactically correct. The problem of syntax checking using a free-form editor is much more difficult since text is inserted and deleted character by character in arbitrary positions. An incremental parser must be built to determine whether a syntactically correct program is being entered, a formidable task in itself. Thus a template-based editor is highly preferable from an implementation standpoint.

4.2 Implementation

Based on the requirements for a suitable syntax to present to the user and the desire to build a template-based editor, *PFLE* is built on top of *Fabform*, a generalized

forms-based editor that provides the user interface for a number of applications in the CAFE[2] system. Appendices B, C, and D provide additional information on *Fabform*.

4.2.1 User's Perspective

The decision to use *Fabform* as the core of *PFLE* provides a number of benefits from the user's perspective. Firstly, *Fabform* provides a user interface that users are very comfortable with. Many applications in the CAFE system, including the menu, machine operation utilities, and machine reservation utility, use *Fabform* as the user interface. Since users are already familiar with the basic commands for *Fabform*, a minimal amount of effort is required to learn to use *PFLE*. Even new users will find *PFLE* easy to learn to use since the commands resemble those of Emacs, one of the most popular text editors. Refer to Appendices A and B for commands supported by *PFLE* and *Fabform*, respectively.

Secondly, since *Fabform* is a forms-based editor, the user deals with a syntax that is represented as a form. Thus the user has the illusion of filling out a form rather than programming in a computer language. Lab users in the Integrated Circuits Lab are very comfortable filling out forms. The machine reservation program presents a form to be filled out for reserving use of a machine for a period of time. Forms are also used to specify the treatment parameters for a batch of wafers destined for the ion implanter. Thus specifying a process flow is not unlike requesting a treatment in the ion implanter – the corresponding form is just much longer. While some non-programmers find the idea of programming in a computer language (or the process flow language for that matter) somewhat frightening, they tend to be very comfortable with the idea of filling out forms.

Figures 4-1, 4-2, 4-3, and 4-4 show the representations of the process flow displayed by *PFLE* corresponding to the flow language descriptions illustrated in fig-

[2]Computer Aided Fabrication Environment

ures 3-1, 3-2, 3-3, and 3-4. As mentioned previous , the distinction between operations and sequences is blurred when presenting the flow to the user. The *furnace-rampup-treatment* sequence illustrated in figure 4-3 appears to be an operation that has a single slot, *:treatment* (in this case, *PFLE* has determined that the sequence can be used only in a *:treatment* slot).

4.2.2 Implementor's Perspective

The advantages of building *PFLE* on top of *Fabform* are even greater from the implementor's perspective.

4.2.2.1 Simplified Implementation

Fabform handles all of the low-level user interaction, freeing *PFLE* from the details of managing information on the screen. Thus the implementation of *PFLE* can be devoted primarily to higher level problems such as syntax and semantic checking, significantly reducing development time.[3]

Since most of the low-level interaction is decoupled from *PFLE*, the majority of the implementation of *PFLE* can be reused if we later discover that *Fabform* is not the most suitable low-level interface. In the meantime, we have a fully functional flow language editor which we can use to evaluate the validity of the higher level functions of *PFLE* (e.g. the syntax and semantic checking, higher level view of the flow, etc.).

4.2.2.2 Syntax Checking

Fabform provides the core of a template-based editor, simplifying the task of syntax checking. An application built on top of *Fabform* sees an event-loop model of

[3] *Fabform* has been developed and refined over the past 16 months. Building a low-level interface for *PFLE* from scratch would have required several weeks of effort and not been nearly as refined as *Fabform*.

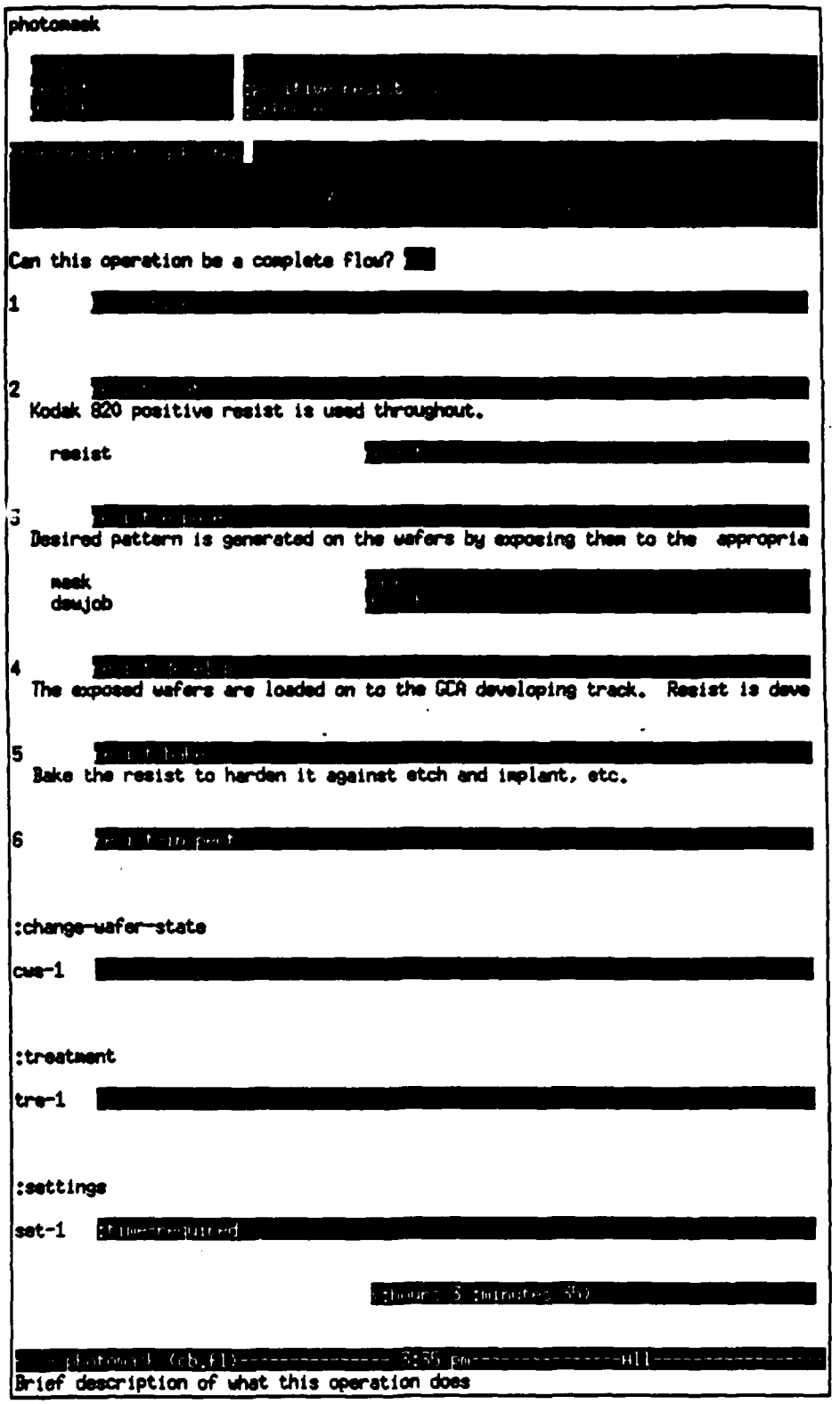

Figure 4-1: *PFLE* representation of *photomask*

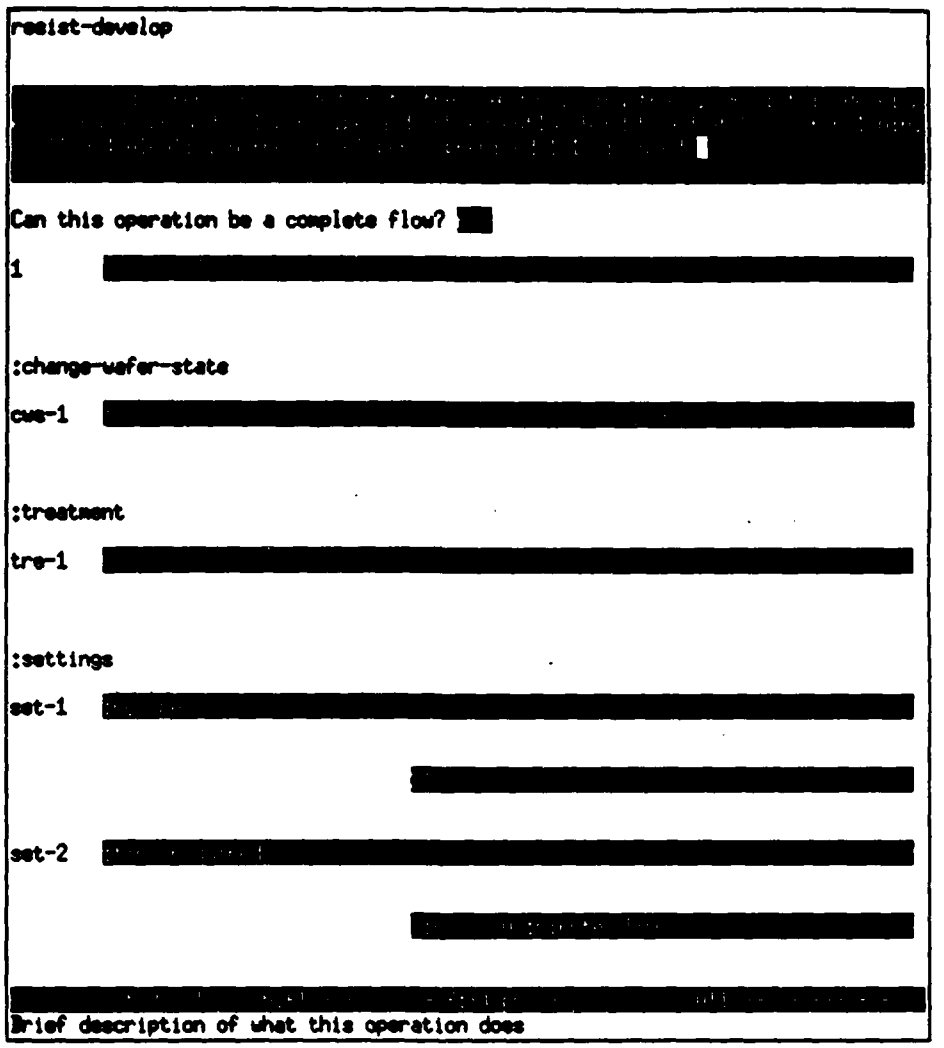

Figure 4-2: *PFLE* representation of *resist-develop*

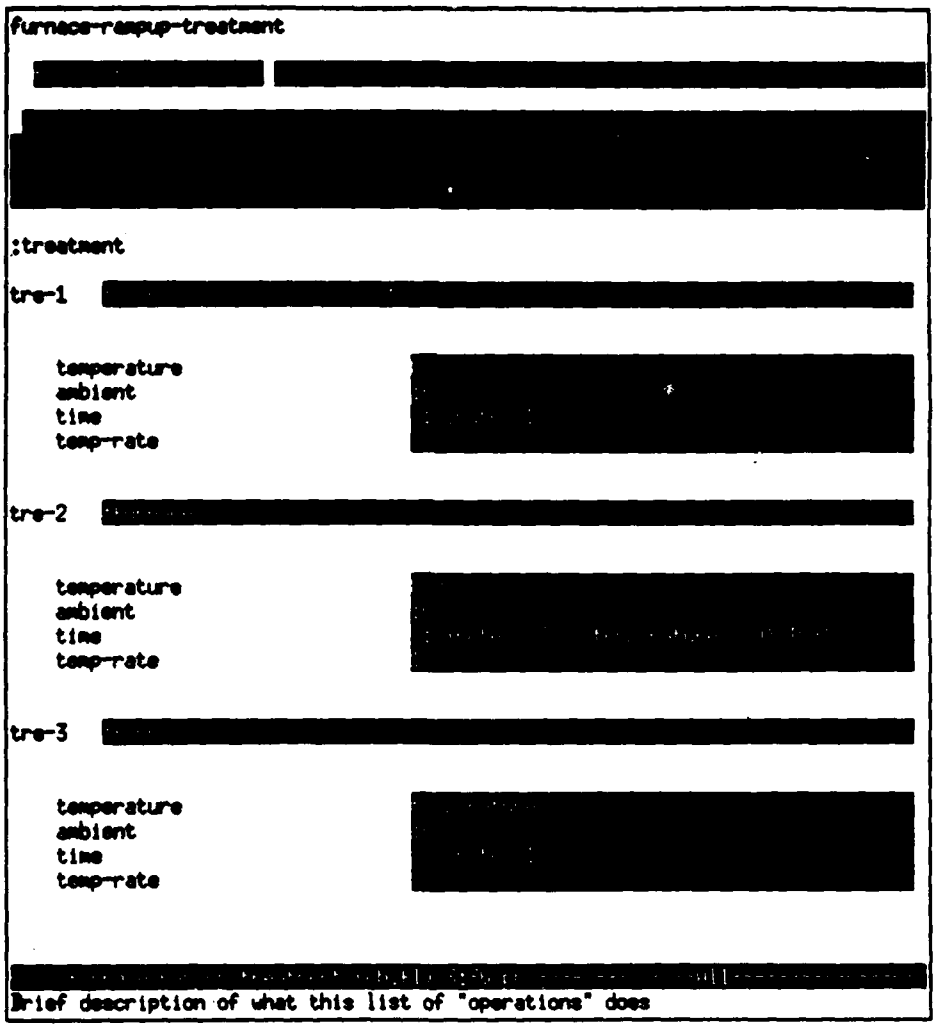

Figure 4-3: *PFLE* representation of *furnace-rampup-treatment*

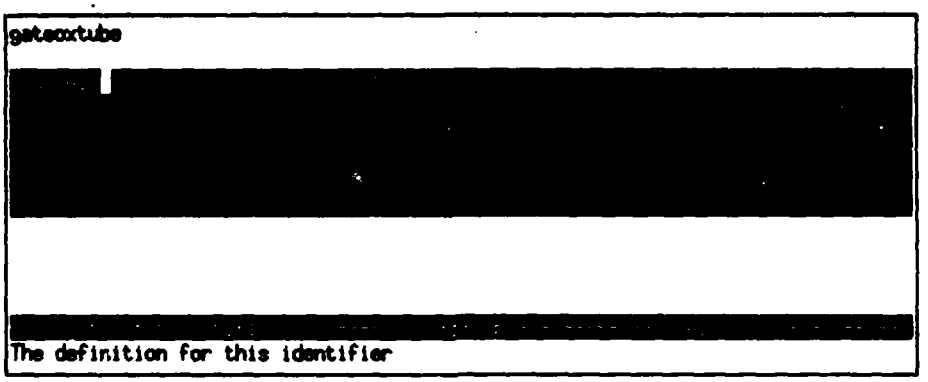

Figure 4-4: *PFLE* representation of *gateoxtube*

computation. It sets up an initial description of a form and hands it to *Fabform*. Whenever the user does something of interest to the application (such as entering a value in a field of the form or pressing a particular keystroke), *Fabform* executes a procedure specified by the application. These procedures can query the user, change values of slots in the form, add or delete fields from the form, etc. The event-loop continues until the user exits *Fabform* and complete control returns to the application. The application may then examine the final state of the form and take the appropriate actions.

Thus, *PFLE* creates a form that corresponds to an object in the flow language and hands it to *Fabform*. *Fabform*, in turn, calls functions defined by *PFLE* whenever the user fills in a field (such as the name of an operation to use) or when the user enters certain keystrokes (such as *control-X control-I* to insert additional fields into the form). When the user exits *Fabform*, *PFLE* reads the description of the resulting form and creates the corresponding flow language object. The problem of syntax checking can be reduced to two subproblems:

1. ensuring that the form always contains a set of fields that correspond to a syntactically correct object in the process flow language

2. ensuring that the fields in the form are filled in with entries that are syntactically correct

The second subproblem is handled completely by *Fabform* since it ensures that a form is filled out correctly.[4]

Thus *PFLE* must ensure that a form contains a syntactically correct collection of fields. The definition of a form is, in effect, the definition of the syntax. Consider the form shown in figure 4-1, for example, and compare it to the corresponding flow object illustrated in figure 3-1. The fields at the top of the form correspond to the

[4]Each field in a form can be restricted to contain only certain types of entries, e.g. integers, floating-point numbers, dates, lisp expressions, etc. *Fabform* ensures that the proper type of value is entered into a field and will not allow an incorrectly filled form to be saved.

declaration of the parameters for the operation; the fields in the left column contain the names of the parameters while those in the right column contain the optional default values of the parameters. The large field below the parameter names contains the optional comment. The remainder of the form contains fields for entering references to operations, sequences, and primitives (all of which are presented as references to parameterized operations). When a parameterized operation is referenced, additional fields are provided for specifying the values to be given to the parameters (the reference to *resist-coat* provides an example).

Once *PFLE* creates an initial form corresponding to a flow language object, it must ensure that fields are inserted or deleted from the form in such a way that a syntactically correct flow language object can be built from the resulting form. Thus, when adding or deleting parameters to an operation, a pair of fields (corresponding to the name and default value of the parameter) is inserted or deleted simultaneously. When adding or deleting a reference to an operation, fields corresponding to the values of all the parameters of the operation are also added or deleted simultaneously. For example, if we added a reference to *photomask* in some other operation, fields would be inserted into the form corresponding to the values of the three parameters of the operation: *mask*, *resist*, and *dswjob*. By always inserting and deleting groups of fields that correspond to syntactic blocks in the flow language, *PFLE* ensures that the resulting form always corresponds to a syntactically correct flow language object.

This practice of inserting fields corresponding to the values of parameters to an operation when an operation reference is inserted provides a useful byproduct: users do not have to memorize the names of parameters to operations. For example, if the user enters a reference to *photomask*, fields are immediately created for the values of its parameters. The names of the parameters are displayed to the left of the fields which the values will be entered into. Thus the user doesn't have to know that *mask*, *resist*, and *dswjob* are the names of the parameters to *photomask*. Even

if he does know the names of the parameters, the user saves a great deal of typing by having the parameter names inserted into the form for him. If he were using a conventional text editor, the user would have had to manually enter the keywords corresponding to the names of the parameters.

Chapter 5

Semantic Checking

The previous chapter described the syntax checking, or syntax enforcing, capabilities of the process flow editor. Syntax enforcement alone, however, will not allow *PFLE* to detect the majority of coding errors as they are made. Some form of semantic checking is required if *PFLE* is to be capable of significantly reducing the time required to correctly code a process flow.

Some of the most commonly made errors, by novices and experienced programmers alike, are the result of mistyping. A syntax checker may detect some typographical mistakes (such as entering an extra parenthesis, etc.), but a more common occurence is a typing mistake that results in misspelling the name of an identifier. The resulting segment of code is syntactically correct, but it obviously does not do what the programmer intended. A semantic checker could prove very useful in detecting errors like these, though even the most advanced semantic checker could be fooled if the misspelled name belonged to another identifier that could be used in the same context. We will examine a few other types of errors that are commonly made which can be detected fairly easily with the aid of a semantic checker.

An interesting characteristic of the semantic checking mechanisms of *PFLE* is that they attempt not only to detect semantic errors but also help the user correct them. Conventional semantic checkers, such as those incorporated into compilers

or utilities like lint, are dedicated to detecting errors and identifying their location. While identifying errors can be very helpful, novices may not know how to correct the problem. Whenever possible, *PFLE* provides a set of options for the most reasonable way to correct the error that was just detected. The following sections describe the semantic checking abilities of *PFLE*, as well as the types of options *PFLE* provides to the user when it detects the errors.

5.1 Parameter Values in Operation References

The view of the flow presented to the user consists of two types of objects: definitions and optionally parameterized operations. The body of an operation may reference other operations, each of which may require values to be specified for parameters to the operations. These operations may, in fact, map to sequences or primitives in the process flow language. Thus semantic checking of parameters to operation references also involves checking of parameters to references to sequences and primitives.

5.1.1 Number of Parameter Values

Since parameter values are passed by keyword in the process flow language and each parameter has a default value (NIL if the default is not explicitly specified), the number of parameter values specified in an operation reference can vary for any given operation. For example, a reference to *photomask* may specify values for all three parameters or may just specify a value for *mask* and use the default values for *resist* and *dswjob*. Thus checking for the number of parameters in an operation reference does not seem to be a worthwhile venture.

A more appropriate semantic check is to ensure that the keywords preceding the values of the parameters match the names of the parameters to the operation. *PFLE* converts this semantic checking problem into a syntax enforcement problem.

Whenever the name of an operation is specified in an operation reference, *PFLE* immediately creates fields into which the values of the parameters to the operation may be entered. The names of the parameters are displayed next to the fields for the values, and *PFLE* translates the forms-based description of the operation reference into its flow language counterpart. The user never has to enter the keywords corresponding to the names of parameters to the operation – *PFLE* automatically inserts the correct keywords for values that are specified (not left blank in the form).

5.1.2 Validity of Parameter Values

Assuming the value for a parameter is syntactically correct, it may still be invalid for one of two reasons:

- It is of the wrong type.
- It is undefined in the current context.

Since the process flow language is essentially type-less, checking the type of a value cannot be performed in general. Thus *PFLE* does not perform any type-checking of values of parameters. Chapter 7, however, discusses the feasibility of adding this feature to *PFLE* in a future implementation.

Checking whether a value is defined in the current context is much more feasible. In general, the value specified for a parameter can be an arbitrary Lisp expression, and the value of the expression would be undefined if the value of any subexpression within the expression was undefined. *PFLE*, however, currently only checks whether identifiers are defined in the current context. Since the process flow language is lexically scoped, checking the validity of an identifier is a fairly simple problem.

When *PFLE* detects an attempt to use an undefined identifier, it displays a menu similar to the one shown in figure 5-1. Rather than simply indicating an error, *PFLE* provides a set of options for correcting the error. In this case, the user is given the option to:

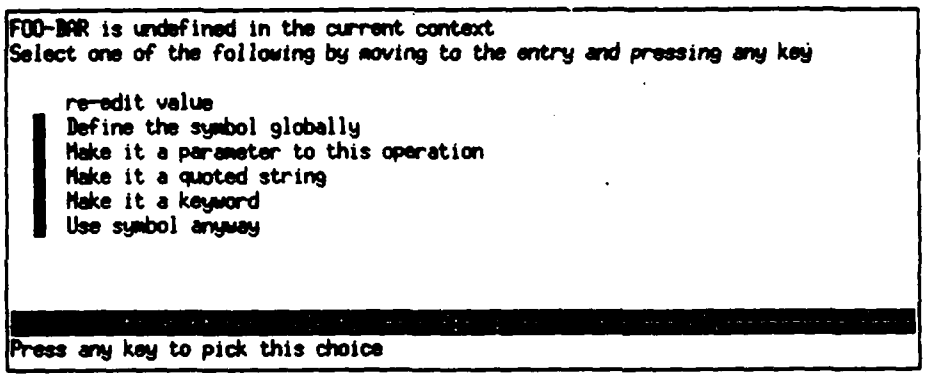

Figure 5-1: Choices available upon discovery of an undefined identifier

- Re-edit the value. In most cases, the user probably simply mistyped the name of the identifier and needs to correct his typing error.

- Define the identifier globally. The user may have forgotten to define the identifier in the global namespace (using a **definition**). If this option is chosen, *PFLE* will first let the user edit the (newly created) definition for the identifier and then return to the the object that was being edited when the error was detected.

- Make the identifier a parameter to the operation. The user may have forgotten to declare the identifier as a parameter to the operation. Rather than making the user move around the form and enter a sequence of keystrokes, choosing this option quickly adds the identifier to the set of parameters for the operation.

- Make the identifier a quoted string. The user may have forgotten to put quotation marks around a value that was meant to be a string (e.g. entering nanospec instead of "nanospec").

- Make the identifier a keyword. The user may have forgotten to prefix the identifier with a colon to indicate that it is a predefined constant (e.g. entering boron instead of :boron).

- Use the identifier anyway. The philosophy behind the design of *PFLE* is that the user always knows best. Even if *PFLE* thinks the user is making a mistake, it allows the user to use an undefined identifier if he insists on using it after being warned. Perhaps the user is going to merge this process flow with another one in which the identifier is defined.

5.2 Type Checking Within Operations

Although the process flow language is type-less as far as the values of parameters to operations are concerned, it is strongly typed in terms of what objects may appear within the slots of an operation. An operation may be composed of multiple slots and references to other operations. Any particular slot may only contain references to primitives or sequences of primitives appropriate for that type of slot. Similarly, all references to other operations must truly refer to operations (in terms of the definition of operations in the underlying flow language, not in terms of the model presented by *PFLE*). The references may not be to primitives or sequences of primitives.

The semantic checking subsystem of *PFLE* is responsible for ensuring that only the proper types of objects are referenced in the various parts of an operation. If the user attempts to reference an object of the wrong type, *PFLE* indicates that the object cannot be used in that position in the flow. If, on the other hand, the object does not exist, *PFLE* offers to create an object of the proper type. If the user decides to create the object, *PFLE* allows him to edit the newly created object (to define the parameters, for example) and then returns to the object that was originally being edited.

5.3 Type Checking Within Sequences

Type checking within sequences is very similar to type checking within operations. If the type of a sequence is known, then the type checking within the sequence is identical to type checking within an individual slot in an operation. If, on the other hand, the type of the sequence is unknown, no type checking is performed within the sequence until the type of the sequence can be inferred.

5.4 Type Inference Mechanisms

Some form of type inference mechanism is required to determine the type of a given object. Without the type inference mechanism, it would be impossible to perform the type checking within operations and sequences. Two type inference mechanisms are used in *PFLE*.

The type of an operation is simply "operation," and the type of a primitive can be determined by using information that defines the primitives for each type of slot within an operation. The type of a sequence, on the other hand, can be determined in one of two ways:

- A sequence inherits its type from the objects that are contained within the sequence.
- If a sequence is used within a slot of an operation or within a sequence of a known type, then the sequence must be of the type that would allow it to be used in that context.

The two separate inference mechanisms allow the type of all sequences in any flow that is being defined via one of the standard design methodologies.

5.4.1 Design Methodologies

The issue of design methodologies can play a major role in the development of an intelligent editor. Some editors support only top-down design, some only bottom-up. The type-inference mechanisms of *PFLE* are designed in such a way that *PFLE* supports both top-down and bottom-up design equally well.

5.4.1.1 Top-Down Design

A top-down design methodology is one in which the highest level of abstraction is defined first, using only specifications of the next lowest level as a building block.

When defining a process flow using a top-down strategy, a process engineer would begin by first defining the entire flow in terms of abstract operations (e.g. the top level CMOS baseline flow is defined in terms of abstract operations such as ***stress-relief-oxidation*** and ***lpcvd-nitride-deposition***). The abstract operations are then defined in terms of lower level operations until the entire flow is eventually defined in terms of existing operations or primitives.

PFLE supports top-down design in a very natural manner. When a reference is made to an operation that does not exist, *PFLE* offers to create that operation. If the user decides to create the operation (which he must do if he wants to reference the operation), he only needs to declare the parameters for the newly created operation. There is no need to actually define the body of the new operation at this point. Simply defining the parameters of the operation supplies enough information for *PFLE* to provide full syntax and semantic checking when references are made to that operation. The user can later re-edit the new operation to define its body.

5.4.1.2 Bottom-Up Design

A bottom-up design methodology is one in which the first level of abstraction is defined in terms of primitives. Each succeeding level of abstraction is defined in terms of the operations already defined on the previous level of abstraction. While a top-down design is preferable for creating a process flow from scratch, a bottom-up approach can be better suited for defining a flow based on a set of predefined operations.

To facilitate bottom-up design, *PFLE* provides two menu feature that provide a list of operations that may be referenced at a particular location in the flow. The first menu function simply provides a list of all the operations that can be legally referenced at a given location in the flow. The other menu function allows the user to place a restriction on the objects that should be listed. For example, figure 5-2 shows the menu produced in response to the question "What can I place in this

:treatment slot that contains a reference to *:furnace*?". Thus the process engineer can choose from a set of predefined operations that perform the wafer treatment that he desires.

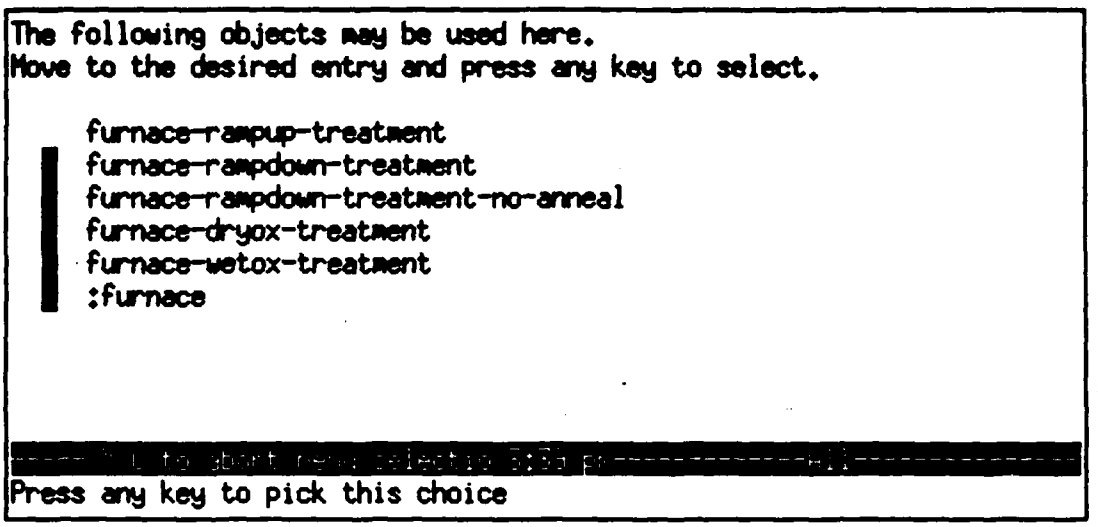

Figure 5-2: Menu of options for *:treatment* slot containing *:furnace*

In some cases, a combination of top-down and bottom-up strategies may be useful. A top-down approach could be used to define the top level of the flow, followed by a bottom-up design of the remainder of the flow to make use of predefined operations. *PFLE* supports both approaches equally and allows both methodologies to be used simultaneously without sacrificing its syntax or semantic checking abilities.

Chapter 6

The Need for Flexibility

Any well-designed software system should be constructed in a manner that allows modifications to be made with minimal effort. The desire to build a flexible system is always present, but, in the case of *PFLE*, a flexible architecture is a requirement. The process flow language is a product of ongoing research, thus it is very likely to evolve in the very near future. *PFLE* must be able to evolve along with the flow language if it is to avoid being a very short-lived editor. The presence of multiple slots within operations also demands flexibility in *PFLE* (as discussed in section 6.2). Finally, an editor designed with flexibility in mind can be enhanced more easily, adding functionality to ease the job of the user coding a process flow.

6.1 Overall Design

Figure 6-1 illustrates the overall design of *PFLE*. The core consists of an internal representation of the process flow and associated information (e.g. type information on objects). The remainder of *PFLE* consists of modules that manipulate information in the core. The modules may also access data external to *PFLE* and reference modules that are either internal or external to *PFLE*. One module, for example, reads a flow coded in the process flow language and creates the corresponding in-

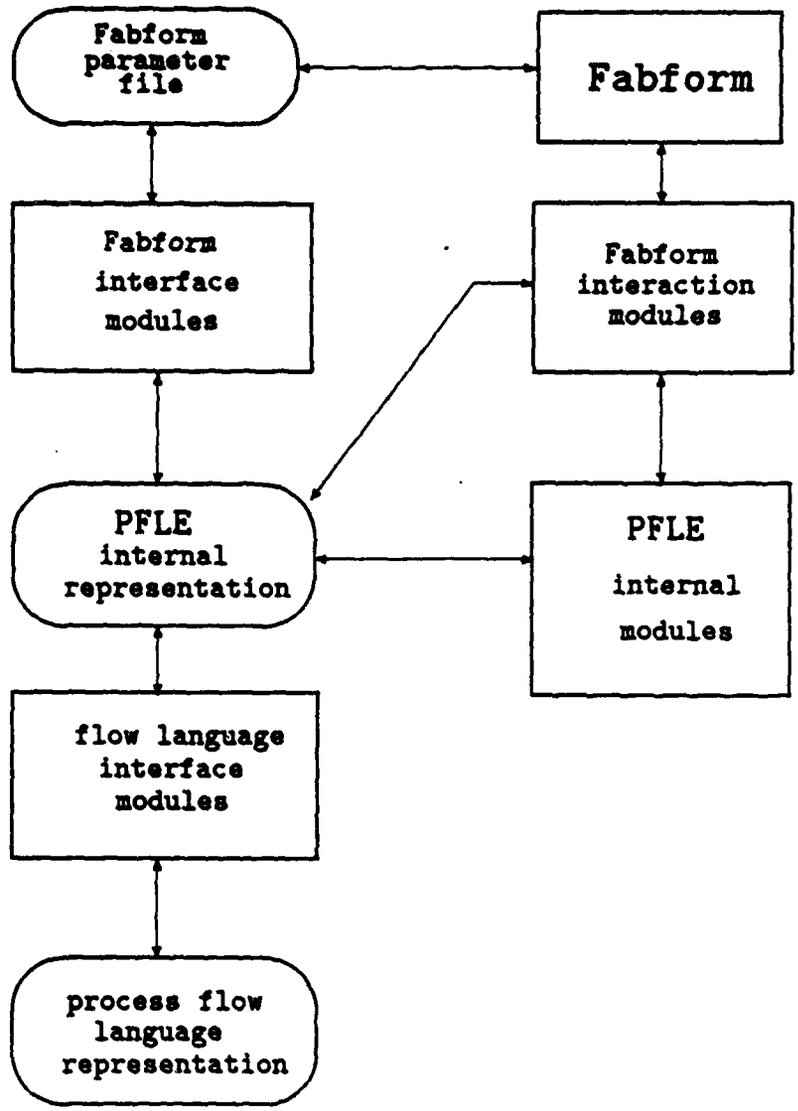

Figure 6-1: Overall Design of *PFLE*

ternal representation. A second module performs the reverse function, thus *PFLE* is well insulated from syntactic changes in the process flow language. Insulation from semantic changes, however, is slightly more difficult, requiring appropriate modifications to the internal representation. All modifications to the internal representation should be made in a manner that least affects the modules accessing the representation.

Another set of modules provides the interface between *PFLE* and *Fabform*. One pair of modules is responsible for converting between the internal representation used by *PFLE* and the parameter files required by *Fabform*. The remainder of the *Fabform*-related modules are called by *Fabform* whenever the user enters a value into a field or presses a particular keystroke. These are the modules that determine what the user has entered in the field and then call the appropriate semantic checking or syntax enforcing modules. Thus the "low-level" user interface can be changed to something other than *Fabform* by replacing the modules that form the interface between *PFLE* and *Fabform*. The majority of the other modules, which perform the semantic checking, etc., and only access the internal flow representation, can be reused. It is very likely that even most of the *Fabform*-related modules could be reused with minimal modifications.

The remainder of the modules within *PFLE* operate only in the internal representation of the flow and may reference other modules within *PFLE*. These modules provide access to objects within the flow, perform type inference and type checking, etc. The modular design allows relatively painless addition and deletion of modules, simplifying modifications to *PFLE*.

6.2 Multiple Slots

The presence of multiple slots within operations provides a unique problem for *PFLE*. A process flow coded in the flow language may be processed by several

different interpreters. Each interpreter may access a different set of slots from the operations in the flow. Thus *PFLE* must possess knowledge about all the possible types of slots that any interpreter may access. When new interpreters are written that access previously unknown slots, *PFLE* must learn how to handle the new slot type.

The problem of providing support for a new slot type is further complicated by the fact that each slot type may have a different syntax, different semantics, and a different set of primitives. The modular design of *PFLE* provides the framework for adding knowledge of a new slot type, however, this framework is still too cumbersome for performing an enhancement that is likely to be repeated several times as new slot types are added. Thus *PFLE* includes an explicit mechanism for adding knowledge of new slot types.

PFLE maintains a list of known slot types and information about the slot. Each entry in the list of slot types contains the following information:

- The slot name, e.g. *:treatment* or *:settings*
- The prefix to use when numbering operation references entered within that slot. Refer to the form representing *resist-develop* in figure 4-2. Entries in the *:change-wafer-state* slot have a prefix of "cws-" while those in the *:treatment* slot have prefix "tre-".
- The function to convert from an entry in the slot (in the internal representation) to its corresponding representation in a *Fabform* parameter file. The internal representation matches the representation of the process flow language, with the exception of some additional fields for *PFLE* generated information.
- The function to convert from a *Fabform* parameter file representation to the internal representation of an entry in the slot.
- The function *Fabform* should execute when the user enters the name of an

operation reference within the slot. This function should then call the appropriate syntax enforcement and semantic checking functions.

- The function that will return a list of known primitives for the slot. This function is only called once when *PFLE* starts. The returned list should be in the internal representation used by *PFLE*.

- The function to lookup primitives for the slot whose names are keywords. Some slot types, e.g. *:settings*, have information on their available in the database, allowing primitives to be individually accessed. The large number of *:settings* primitives also makes it impractical to drag information on all primitives from the database when *PFLE* starts up. Thus the information is pulled from the database as it is required.

Adding a new slot type to *PFLE* requires the addition of an entry into the list of slot types to describe the new slot. Any new functions referenced by the description of the slot will also have to be added. The current implementation of *PFLE* contains knowledge of three slot types: *:change-wafer-state*, *:treatment*, and *:settings*. All three slot types share the functions specified in their descriptions (since they are very similar in syntax and semantics), with the exception of the function to lookup primitives for the *:settings* slot. Thus adding a new slot type that has syntax and semantics similar to existing slot types is very easy. Adding a slot with different syntax or semantics requires a little more work since a few functions to handle the new slot type will have to be defined.

6.3 Adding Functionality

One final consideration in designing *PFLE* is that *PFLE* is intended to be much more than an editor. The syntax and semantic checking abilities of *PFLE* are very powerful tools and certainly simplify the coding of a process flow in the flow

language. As we use *PFLE* to define process flows, however, we are bound to find that we can increase its utility by extending its functionality.

While syntax and semantic checking prove adequate for detecting coding errors in most programming languages, the process flow language, or rather the manner in which it is used, demands additional checking mechanisms. The multiple slots within operations, for example, provide a source of coding errors not found in other languages. Each slot should contain a different view of the same operation. The current implementation of *PFLE* does not check for consistency between the contents of slots (largely because such consistency checking is very difficult to implement). A user could define an operation in which the *:change-wafer-state* slot describes an oxidation step while the *:treatment* slot describes an ion implant treatment. Clearly the operation is incorrectly coded, yet it is syntactically and semantically correct. Other useful features could be automatic checking of flows for conformance to lab policy, sanity checking (e.g. perform a *clean* operation before a *furnace* step), etc.

Another long term goal is to build an expert system for creating process flows. The user should be able to ask the system to "grow 1000 angstroms of oxide" and have the program automatically generate a segment of the flow that grows 1000 angstroms of oxide. *PFLE* should provide a platform on which such an expert system can be built, allowing users to continue using a tool they are familiar with while reaping the benefits of extended functionality.

Chapter 7

Results and Directions for Future Work

7.1 Implementation Status

PFLE is fully operational and supports all of the syntax and semantic checking described in chapters 4 and 5. An area of concern while developing *PFLE* was the validity of the higher level view of the flow (consisting of only operations and definitions) presented to the user. The process flow language was still in a state of flux during the early stages of the development of *PFLE*, thus it was unclear whether the subset of the process flow language chosen as the basis for *PFLE* possessed sufficient expressive power. During the development of *PFLE*, it has been tested continually using a coding of the CMOS baseline process. The view of the flow used by *PFLE* is able to express the entire CMOS baseline flow, and since most of the processes used in the Integrated Circuits Laboratory are derivatives of the CMOS baseline process, *PFLE* should be able to handle most, if not all, of the flows that users will want to define. The model used by *PFLE* may have to evolve as the flow language itself evolves, but at the present time, it appears that the currently implemented model is more than adequate.

Although *PFLE* is designed to be an editor for novices and expert programmers alike, no one is forced to use it to edit process flows. *PFLE* should provide enough hand-holding to lead a novice user through the steps of defining a process flow while avoiding the trap of excessively hampering the expert programmer. Even the most experienced programmers make mistakes, and *PFLE* can detect those mistakes as they are made, thereby improving programmer productivity. The current version of *PFLE* is capable of reading process flows created using other editors. The coding of the CMOS baseline process that was used to test *PFLE* was created using a conventional text editor. Flows edited using *PFLE* can also be edited using other editors since *PFLE* saves flows in the syntax of the process flow language. This ability to use different editors to manipulate a single process flow will become increasingly important in the future as different types of editors are developed. Some users may prefer an editor with a graphical interface, for example, while others may prefer *PFLE*.

PFLE is implemented in Common Lisp to facilitate manipulations on the Lisp-like structures used by the process flow language. *PFLE* is quite portable and is currently running under both Allegro Extended Common Lisp and Kyoto Common Lisp.[1]

7.2 Enhanced Semantic Checking

The semantic checking abilities of *PFLE* are quite extensive, but they still fall short of being able to detect all semantic errors before run-time. There are three major areas in which the semantic checking can be improved:

- Type checking of parameters to primitives

[1] *Fabform* is written in C but provides Lisp interfaces under Allegro Extended Common Lisp and Kyoto Common Lisp. The CAFE system currently runs only under Allegro but is in the process of being ported to Kyoto Common Lisp.

- Type checking in conditional expressions
- Consistency checking after modifying a section of the flow

Currently *PFLE* does not perform any type checking on the values specified for parameters of operations (or primitives). Since the process flow language, like the Lisp it is embedded in, is essentially a typeless language, the type of an object cannot be determined until run-time. The primitives for certain slots, however, may require parameters to be of a particular type. The *:implant* primitive for the *:treatment* slot, for example, requires a parameter called *:element* that must be a keyword specifying the element used as the dopant. Using an integer as the value of the parameter would clearly be wrong, but *PFLE* would not catch that error. Unfortunately *PFLE* does not currently have such type information available. The problem of type-checking parameters is further complicated by the fact that the value of a parameter may be an arbitrary Lisp expression that is evaluated at run-time. Nevertheless, at least a few semantic errors could be detected even if *PFLE* only type-checked parameters whose values were specified as constants.

Conditional expressions were added to *PFLE* partly as an afterthought. They appear only in two "low-level" operations in the CMOS Baseline flow and were initially considered to be one of those rare cases that *PFLE* could not handle. Once these "low-level" operations were implemented, no one would have to define anything similar, so conditionals were thought not to be very important in the first implementation of *PFLE*. Since it appears, however, that conditional expressions may play an increasingly important role in the flow language in the future, minimal support was added to *PFLE* to handle conditional expressions. When an *if* is entered as the name of an operation in an operation reference, fields are added to the form for the predicate, consequent, and alternate. Each of the fields may be filled with an arbitrary Lisp expression, and no semantic checking is done on any of those fields. A future implementation of *PFLE* should provide more complete support for conditional expressions and provide semantic checking for the components of the

conditional.

Checking the flow for consistency after modifying a segment of the flow is an issue that has been deferred in this implementation of *PFLE*. All semantic checks are performed based on the state of the flow while an object is being edited, but the impact of editing the object is not considered. For example, consider a process flow that contains two operations: *foo* and *bar*. Suppose that *foo* takes three parameters, and *bar* references *foo* and specifies values for the three parameters. *foo* is subsequently edited and one of the parameters is deleted, resulting in a definition of *foo* that only has two parameters. The new definition is syntactically and semantically correct, but *bar* is now semantically incorrect because it specifies too many parameters for *foo*. *PFLE* does not currently perform any such consistency checks because a brute-force method for checking consistency would be extremely expensive in terms of computation time. Nevertheless, it is an important issue to consider for future implementations.

7.3 Hiding Lisp Syntax

Chapter 4 describes the view of the flow presented to the user. This view differs significantly from the syntax of the underlying process flow language. It provides a higher level representation that is simpler to understand and hides the Lisp-like syntax of the process flow language that tends to frighten non-programmers. While *PFLE* hides most of the Lisp syntax, it does not hide all of it. Two places where the Lisp syntax still appears are:

- conditional expressions
- values of parameters in operation references

The problems associated with conditional expressions was discussed in the preceding section. The Lisp syntax in the value of a parameter can appear either in the form

of a function application (e.g. "(+ minutes 10)") or in the form of notation for an abstract data type (e.g. "(:hours 2 :minutes 15)"). There doesn't seem to be an easy way to eliminate these expressions without building a method of mapping generic Lisp expressions to a forms-based representation. If the CMOS baseline process is representative of the types of process flows users will be defining, function application expressions are rarely used as values to parameters. The parenthesized notation for abstract data types, e.g. time, is fairly common, but that notation seems like a very easy one for users to learn.[2] It should also be possible to develop a forms-based representation for each of the abstract data types and eliminate parentheses in data type expressions altogether.

7.4 Support for the X Window System

The current version of *PFLE* is based on *Fabform*, which runs only on ASCII terminals. Under the X window system, *PFLE* may be run within an xterm window. *PFLE* provides extended functionality under X by allowing the user to create additional windows so that he may simultaneously view multiple flow language objects. Only a single object may be modified at a time, however, to avoid consistency problems among multiple editing windows.

One area of future work should be additional support for the X window system. It is not clear whether *PFLE* itself should be modified to directly support X or whether X support should be incorporated into *Fabform*. The greatest overall benefit will probably be derived by upgrading *Fabform* and incorporating a coroutining mechanism in *PFLE* to manage multiple windows simultaneously. The job of adding support for X will become much easier as the X toolkit matures and more powerful widgets are developed (such as the fabled xterm widget).

[2] They can probably learn the construct by simply looking at the CMOS baseline flow and coding by example.

7.5 Database Support

PFLE currently accesses the database (via the *Gestalt* functional data model) to obtain information about primitives for the *:settings* slot of an operation. Information on primitives for other slot types is currently read from files because it is not available in the database. As more information is added to the database, *PFLE* should be modified to access that information.

Appendix A

The PFLE User Manual

PFLE Users Manual

Rajeev Jayavant

Revised August 25, 1988

A.1 Introduction

A process flow is the sequence of operations performed on a silicon wafer in order to fabricate integrated circuits. *PFLE* is an intelligent editor designed to facilitate the task of encoding a process flow in the process flow language. It is intended to be used by a wide range of users, from novices to experienced programmers. *PFLE* provides a number of features that assist less experienced programmers, while seasoned programmers will benefit from *PFLE*'s error detecting mechanisms.

Users that have coded process flows using a text editor will immediately notice that *PFLE* presents the flow in a format that is very different from the representation used by the process flow language. The process flow language is embedded in Common Lisp and shares the parenthesized syntax of Lisp. *PFLE* provides a forms-based interface to the process flow language in which each object is represented by a form that the user can edit. In addition, *PFLE* provides a view of the process flow that is composed only of two classes of objects: *definitions* and *operations*. *Definitions* are used for specifying *constants*, e.g. define *gateoxtube* to be the name of the furnace tube in which gate oxide is grown. The remainder of the process flow can then refer to *gateoxtube* rather than the actual name of the furnace tube.

Operations are used to specify the actual sequence of processing steps performed on a wafer. An operation may be parameterized and is defined in terms of other operations (either user-defined or primitives supported by an interpreter). Thus a process flow is defined in terms of a hierarchy of operations – the top level of the hierarchy specifies the most abstract view of the flow while the lowest level of the hierarchy is defined in terms of primitive operations.

Although *PFLE* presents a view of the flow that consists only of *operations* and *definitions*, the process flow language supports a number of other object types, e.g. sequences. *PFLE* attempts to simplify the view of the flow by making all objects in the process flow language "look" like *operations* or *definitions*. Thus there are actually two different types of *operations* in *PFLE*:

- operations with multiple slots [1]
- operations without slots

An operation with multiple slots may only be referenced within another operation with multiple slots and may not be referenced within a slot of an operation. An operation without slots, on the other hand, may be used within a slot of an operation with multiple slots. If the user defines a process flow using a top-down design methodology, *PFLE* will automatically create the proper type of operation (with multiple slots or without). In any case, the semantic checking mechanisms of *PFLE* will prevent incorrect references to operations of the wrong type, thus the user does not have to worry about the different types of operations.

This manual attempts to describe the operation of *PFLE* and how it can be used to create a new process flow or edit an existing flow. It first describes the various commands supported by *PFLE* and then illustrates a sample session in which we create a very simple process flow using *PFLE*. Even the most detailed manual, however, cannot really teach a user how to use a piece of software. Only hands-on experience can provide a true understanding of how something works, thus the user is encouraged to experiment with *PFLE* on a sample process flow while reading this manual.

A.2 Entering and Exiting *PFLE*

PFLE can be started from the Wafer Menu in CAFE. Two options should be provided:

- Edit process flow
- View process flow

If you choose to view a process flow, *PFLE* will operate in "read-only" mode and prevent any modifications to the process flow. *PFLE* will first prompt for the name

[1] The process flow language supports operations with multiple slots for specifying different views of the same processing step. The process flow language itself does not specify which slots are defined and what their semantics are. Each interpreter assigns semantics to slots that it is interested in. The two-stage process model specifies the flow in three levels: *change-wafer-state*, *treatment*, and *settings*. The *change-wafer-state* view specifies the change in the wafer state in abstract terms, e.g. grow 1000 angstroms of oxide. The *treatment* view specifies wafer treatments in terms of temperature, gas concentrations, treatment time, etc. The *settings* view specifies the actual settings of a particular machine that will process the wafer. Equipment models and process simulators can be used to convert from one vew of the flow to another.

of the file containing the process flow. The ".fl" suffix is automatically added if it is not specified. *PFLE* will then read the process flow from the file and display the top-level view of the flow (refer to section A.4 for more information). If the file does not exist, *PFLE* will assume that you want to create a new flow.

To exit *PFLE*, simply enter ctrl-X ctrl-C when the top-level view of the flow is being displayed.[2] If the process flow was modified, *PFLE* will ask whether the updated flow should be written. Answering anything other then "y" will discard any changes made to the flow during the editing session.

A.3 Basic Commands

PFLE relies on *Fabform* to provide the user interface, thus all of the commands supported by *Fabform* are supported by *PFLE*. The basic commands for cursor movement and exiting the editor are analogous to those for Emacs. Please refer to the *User Guide to the Fabform User Interface* for further details. The remainder of this manual will describe commands specific to *PFLE*.

The command set of *PFLE* consists of two subsets:

- commands that are always available
- commands specific to an operating mode of *PFLE*. The three main operating modes of *PFLE* consist of:
  1. display top-level view
  2. edit an operation
  3. edit a definition

The following table briefly describes the subset commands that are available in all operating modes. More detailed descriptions of the commands are available in the remainder of the manual, along with descriptions of commands specific to operating modes.

ctrl-X ctrl-C Exit current screen. This is actually a *Fabform* command, but it of special importance within *PFLE*. *PFLE* invokes *Fabform* each time a new object is edited or viewed. Pressing ctrl-X ctrl-C will terminate the latest

[2] It may be necessary to press ctrl-X ctrl-C multiple times to return to the top-level view of the flow. Refer to the following sections for more information.

invocation of *Fabform* and resume editing the object that was previously being displayed.

ctrl-X ctrl-E Edit the object specified at the current cursor position. This command can be used to quickly walk down the hierarchy of operations by moving to the desired operation reference within an operation and pressing a keystroke to edit the referenced operation.

ctrl-X E Edit an object. *PFLE* will prompt for the name of the object to edit. If the object does not exist, *PFLE* will optionally create the object (after displaying a menu to query the type of object to create).

ctrl-X ctrl-V View the object specified at the current cursor position. Similar to ctrl-X ctrl-E except that the object cannot be accidentally modified.

ctrl-X V View an object. *PFLE* will prompt for the name of the object to view. The object being viewed cannot be accidentally modified.

ctrl-X 2 Copy window. If *PFLE* is being run under the X window system, a new window containing the contents of the current window can be created. The new window can only be used to view the object being displayed, and only *Fabform* commands will be available in the new window (*PFLE* commands will not work in the new window). All commands will work in their usual manner in the original window.

ESC N Move cursor to the next operation reference. A repeat count may be specified using ctrl-U.

ESC P Move cursor to the previous operation reference. A repeat count may be specified using ctrl-U.

A.4 The Top-Level View

The top-level view of the flow displays the names of all of the objects in the process flow. The commands described above can then be used to edit or view any of the objects in the process flow, as well as to create new objects. The top-level provides a number of additional commands for deleting and creating objects. The following table describes commands specific to the top-level view.

ctrl-X ctrl-D Delete object. Deletes the object at the current cursor position after confirming the user's intentions. If a repeat count of N is specified using ctrl-U, *PFLE* will confirm the deletion request for each of the N objects beginning with the object under the cursor. *PFLE* does not currently perform any consistency checking to see whether the deleted object is required by any

other object in the flow, thus the user should be careful about what he or she deletes.

ctrl-X ctrl-I Insert flow. *PFLE* will prompt for the name of a file from which to read a flow. Any objects which are defined in the new flow but are not defined in the flow being edited are appended to the flow being edited. This command is useful for defining a new flow using pieces of an existing flow. After the new flow is read, some type inference is performed and a new top-level view is created.

ctrl-X ctrl-R or ctrl-X R Recompute the top-level view. When objects are created using **ctrl-X ctrl-E**, the top-level view is not automatically updated to include the new object. This command can be used to force recomputation of the top-level view.

ESC T Redo type inference. When a process flow is inserted, *PFLE* performs some incremental type inference to determine the types of the newly included objects. In some cases, this incremental type inference may be insufficient to provide the information required for *PFLE* to perform semantic checking. Pressing **ESC T** will force *PFLE* to discard all type information and perform full type inference again. By only performing incremental type inference when inserting flows, *PFLE* reduces the time required to insert multiple flows while preserving full functionality in most cases.

A.5 Editing Definitions

A definition can be edited by using any of the commands for editing or viewing an object. In addition, a definition can also be created (and edited) by referencing the undefined object within an operation (refer to section A.6.3.2 for details).

The form representing a definition consists of only one editable field - the value being assigned to the object. The value may be any arbitrary Lisp expression. *PFLE* does not currently perform any semantic checking on the use of definitions, thus the user should be careful to ensure that the values assigned to objects are semantically valid.

PFLE does not support any special commands while editing a definition. Once the field containing has been filled in, press **ctrl-X ctrl-C** to exit the definition screen and save the new definition. *PFLE* will then return to editing the object that was previously being edited, or to the top-level view if no other object was being edited.

A.6 Editing Operations

Operations, like definitions, can be edited using any of the commands for editing or viewing objects in the process flow. New operations can also be created by referencing the undefined operation within another operation. Since *PFLE* can create unknown operations when they are referenced, defining a process flow using a top-down design methodology is greatly simplified.

The form representing an operation contains many different types of fields corresponding to the various components of an operation:

- parameters for the operation
- a comment describing the function of the operation
- references to other operations and the values of parameters to the other operations
- each slot within an operation with multiple slots may contain references to other operations

PFLE provides commands for creating, deleting, and specifying values for the various types of fields, thereby providing the means for editing operations. The actions of the *PFLE* commands for editing operations, however, can vary slightly depending on which field the cursor is positioned upon. For example, ESC ctrl-I always inserts a field before the field the cursor is positioned upon. The type of field that is inserted, however, depends on the field that the cursor is positioned upon when the command is executed. The basic commands supported while editing an operation appear in the table below. The following sections describe the variations their actions depending on the position of the cursor.

ctrl-X ctrl-D Delete the field the cursor is currently on.

ctrl-X ctrl-I or ctrl-X I Insert field after the field the cursor is currently on. **ctrl-U** may be used to specify a repeat count.

ctrl-X ctrl-M or ctrl-X M Choose operation to reference from a menu

ctrl-X ctrl-U or ctrl-X U Create an unnamed operation within the current operation. Refer to section A.6.3.1 for details.

ESC I Insert field before the field the cursor is currently on. **ctrl-U** may be used to specify a repeat count.

ESC M Choose operation to reference from a menu containing operations which reference a specified object. *PFLE* will query the name of the desired object.

A.6.1 Parameterized Operations

Any operation defined using *PFLE* may be parameterized by simply specifying the parameters to the operation when the operation is defined. Each parameter is represented by two fields. The leftmost field is used to specify the name of the parameter. The rightmost field may be used to specify the default value for the parameter. Whenever a parameterized operation is referenced, values may be specified for any of the parameters. If the value for a particular parameter is not specified, its default value is used (or NIL if no default is specified).

The fields for specifying parameter names and default values appear at the top of the form, if any parameters are specified. Additional fields may be created by moving onto an existing parameter field or onto the comment field (see below) and pressing one of the keystrokes for inserting a field.

A.6.2 Descriptive Comment

Each operation may optionally contain a comment describing the purpose of the operation. The field corresponding to the comment appears below the fields for specifying parameters, or at the top of the form if no parameters have been specified. The comment may be any string of characters up to the length of the field. *PFLE* displays the start of the comment of an operation wherever the operation is referenced within another operation, thus comments are a very useful documentation tool for not only the operation containing the comment but for any operations that reference it.

A.6.3 References to Other Operations

The main body of an operation consists of one or more sets of references to other operations. An operation with multiple slots contains a set of references for each slot in addition to a set for references to other operations with multiple slots. An operation without slots contains a single set of references to other operations.

Each operation reference is represented by a a field containing the name of the operation being referenced, as well as fields for specifying values for parameters of the operation being referenced. The fields are accompanied by a set of associated information:

- A sequence *number* indicating the position of this operation reference within the set of references is displayed to the left of the operation name. If the

reference is to an operation with multiple slots, the sequence number will be an integer. If the reference is to an operation with a single slot, the sequence *number* will be an integer with a prefix corresponding to the name of the slot that the operation may be referenced in.

- The first line of the comment describing the operation being referenced is displayed below the operation name, if the comment is defined.
- The names of the parameter to the operation are displayed to the left of the fields in which the values of the parameters may be entered. There is no need to remember the order of parameters to an operation since the name of the parameter is always displayed next to the value being specified.

Operation references are inserted and deleted as monolithic units by moving the cursor onto any field within an operation reference and pressing a keystroke for inserting or deleting a field. When an operation reference is inserted, *PFLE* inserts an empty field for the name of the operation and provides a step number with the same prefix as the field which the cursor was on when the insert command was given. In an operation with multiple slots, *PFLE* ensures that at least one field for an operation reference remains in each of the slots at all times, even though that field may be empty. Thus it will always be possible to create an operation reference in any slot.

A.6.3.1 Inserting Operation References

Inserting an operation reference is actually a two-step process. The first step is to create a blank field for the reference. As described in the previous section, a blank field can be created by moving to a an existing operation reference and pressing one of the keystrokes for inserting a field. *PFLE* initially provides one blank field per slot for an operation reference, so there may not be a need to create a new field.

The second step is to actually specify the operation to reference. This can be done in a number of ways:

- Enter the name of the operation into the appropriate field by typing the name into the field and pressing RETURN or moving the cursor to another field. If the specified operation doesn't exist, *PFLE* will offer to create it. If a new operation is created, *PFLE* will begin editing the newly created operation. It is important to define all parameters for the newly created operation before returning to the original operation being edited. *PFLE* requires specifications of parameters to perform semantic checking of references to operations.

- Use one of the menu functions to choose the operation to reference. Both menu functions will display a menu of operations that may be referenced in the current context from which the desired operation may be chosen. The menu function invoked by ESC M restricts the menu of operations to those which contain references to an object specified by the user. Both menu functions can be useful when defining a flow based on previously defined operations (such as those copied from another process flow).
- Define an unnamed operation. An unnamed operation is an operation with multiple slots that is contained entirely within the operation that references it. Unnamed operations do not appear in the top-level view of the flow. The only way to edit an unnamed operation is to first edit the operation that references it, move the cursor to the reference to the unnamed operation, and then press **ctrl-X ctrl-E**. Unnamed operations cannot be parameterized, but they may reference the values of the parameters of the enclosing operation.

Once the operation has been specified, *PFLE* replaces the fields for the empty operation reference with fields applicable to the specified operation. New fields are created for specifying values for the parameters to the operation and the comment describing the operation is displayed.

A.6.3.2 Specifying Values of Parameters

The value of a parameter to an operation may be any arbitrary Lisp expression. *PFLE* does not perform semantic checking on arbitrary Lisp expressions, thus the user should be careful to ensure that the values specified for parameters to operations are semantically correct.

If the value specified is an identifier, however, *PFLE* will ensure that the identifier is valid in the current context. If the identifier is undefined, *PFLE* provides a menu of options for correcting the problem rather than simply complaining about illegally using an identifier. The choices provided to the user are summarized below.

- re-edit the value of the parameter. In most cases, the error will simply be caused by mistyping the name of the identifier, thus the first menu option allows the user to correct his typing mistake.
- define the identifier globally. Perhaps the user forgot to define the identifier using a definition. Choosing this option allows the user to create a new definition and then return to editing the operation he was editing.
- make the identifier a parameter to the operation. Adding a parameter would normally take several keystrokes, yet it can be done with a single keystroke

via this menu option.

- make the identifier a keyword. Perhaps the user forgot to type the colon.
- make the identifier a quoted string. It's easy to accidentally type nanospec instead of "nanospec".
- use the identifier anyway. The user may know what he is doing and will define the identifier at a later time.

A.7 Creating a Simple Process Flow

This section describes how a very simple process flow can be created using *PFLE*. The actual flow that is defined is not important since since this example will illustrate all of the concepts necessary for defining a much more complex process flow. The particular flow that will be created is one in which we begin with a bare silicon wafer, grow a thick field oxide, and then pattern an active area region in the oxide.

Begin by selecting the "Edit process flow" entry in the Wafer menu in CAFE. When prompted for the name of the flow to edit, specify **simple**, forcing *PFLE* to look for the file "simple.fl". Since the file doesn't exist, the top-level view of the flow will not contain any objects. In order to make our job a little easier, we'll define our simple process flow in terms of operations defined in an existing process flow (such as the CMOS Baseline process flow). First we'll insert the contents of the existing flow by pressing **ctrl-X ctrl-I** and specifying the name of the file containing the existing flow. The top-level view should now be full of objects. We can also delete any unneeded operations using **ctrl-X ctrl-D**, but at this point we may not know exactly what we need and what can be deleted.

Now that we have all of the underlying operations we need, we can begin our real job. In most cases, designing a new process flow is much easier using a top-down approach, thus that is the path we shall take. We begin by defining the top level of our operation hierarchy by creating the operation *simple*. Press **ctrl-X ctrl-E** to edit an object and specify **simple** as the name of the object to edit. *PFLE* will then display a menu from which to select the type of object to edit, as illustrated in figure A-1. Select the option that creates "a flow or operation with n slots".

PFLE will then present a screen representing *simple*, an operation with multiple slots. Enter the descriptive comment, answer "yes" to the question asking whether the operation can be a complete process flow, and move the cursor to the field next to the number "1". This is the field for entering the first reference to another operation with multiple slots. We would like to define *simple* in terms of two operations, *grow-field-oxide* and *pattern-active-area*, so we will need another slot for

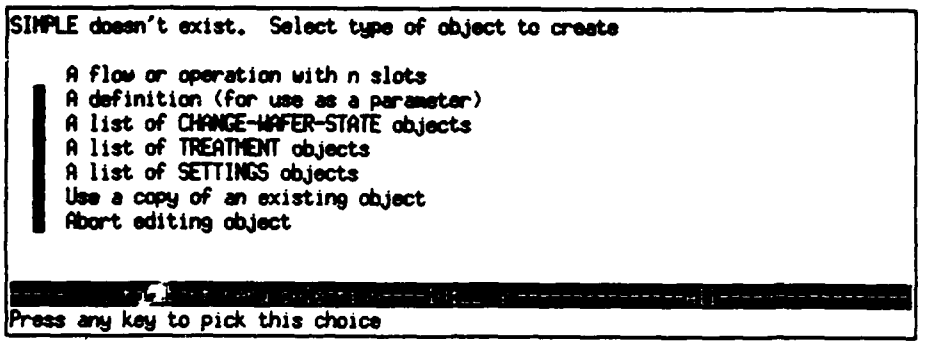

Figure A-1: Menu for selecting type of object to create

an operation reference. Press **ctrl-X ctrl-I** to create a second field for entering an operation reference. The screen should now resemble the one in figure A-2.

Enter "`grow-field-oxide`" in the field next to the "1" and press RETURN. *PFLE* should complain that *grow-field-oxide* is not defined and offer to create it for you. Answer "y" to begin editing *grow-field-oxide*. Notice that *PFLE* did not ask what type of object to create since it knows only a single type of object can be referenced in that particular context. Since we are doing a top-down design, it is not necessary to define the body of *grow-field-oxidation* at this point. We must, however, define any parameters that are required, and it is always a good idea to fill in the descriptive comment to aid in documentation. Since *grow-field-oxide* does not require any parameters, simply fill out the comment field and press **ctrl-X ctrl-C** to return to *simple*. The screen should now resemble the one illustrated in figure A-3.

Similarly, enter "`pattern-active-area`" in the field next to the "2" and create the new operation. Let's suppose that *pattern-active-area* requires a single parameter, *mask*, thus we must define the parameter before returning to *simple*. Move the cursor to the comment field and press **ctrl-X ctrl-I** to insert fields for defining a parameter. Enter the name of the parameter in the leftmost field, the default value for the rightmost field, and description of the operation in the comment field. The screen should now resemble the one shown in figure A-4. We can then return once again to *simple* and complete the top level description of the flow. Figure A-5 illustrates the completed top level.

The steps illustrated for defining the top level of the simple flow can be used repeat-

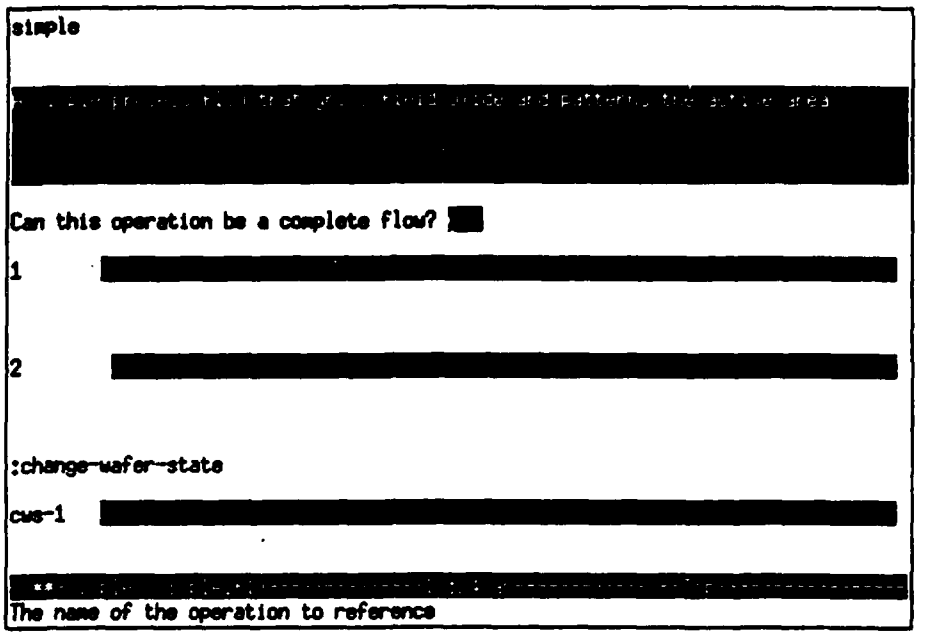

Figure A-2: *simple* with fields for two references

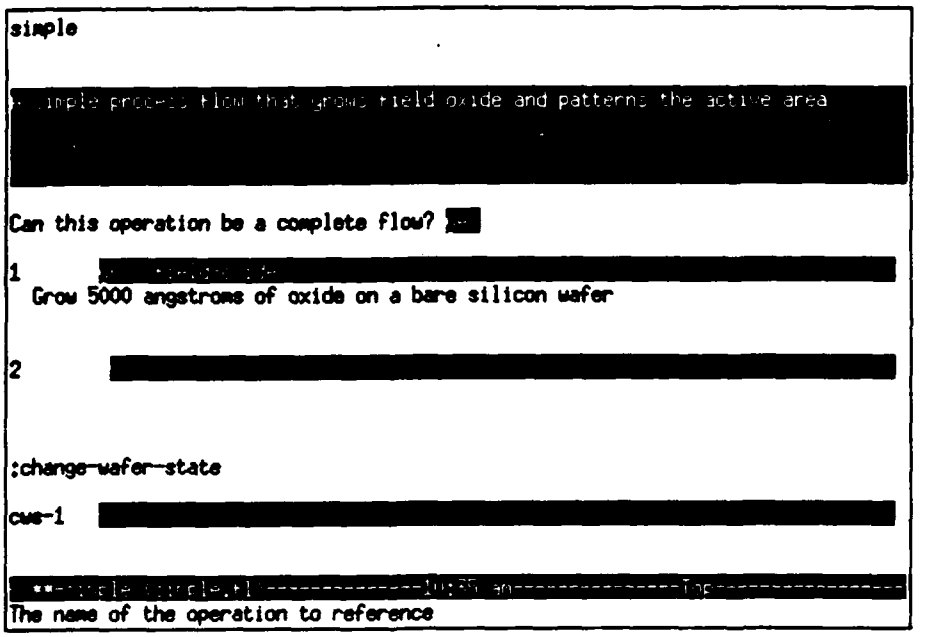

Figure A-3: *simple* with reference to *grow-field-oxide*

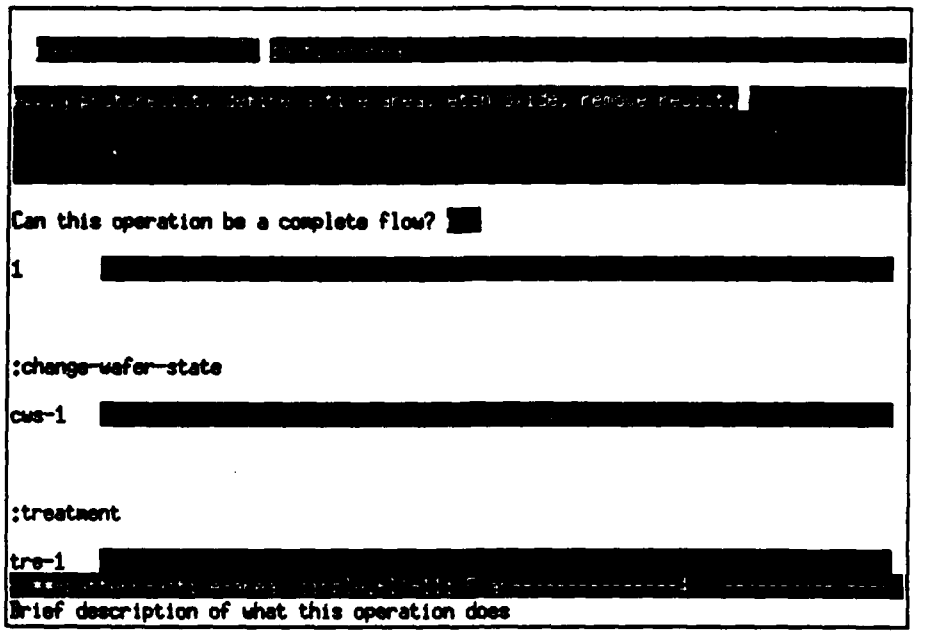

Figure A-4: *pattern-active-area* with parameter defined

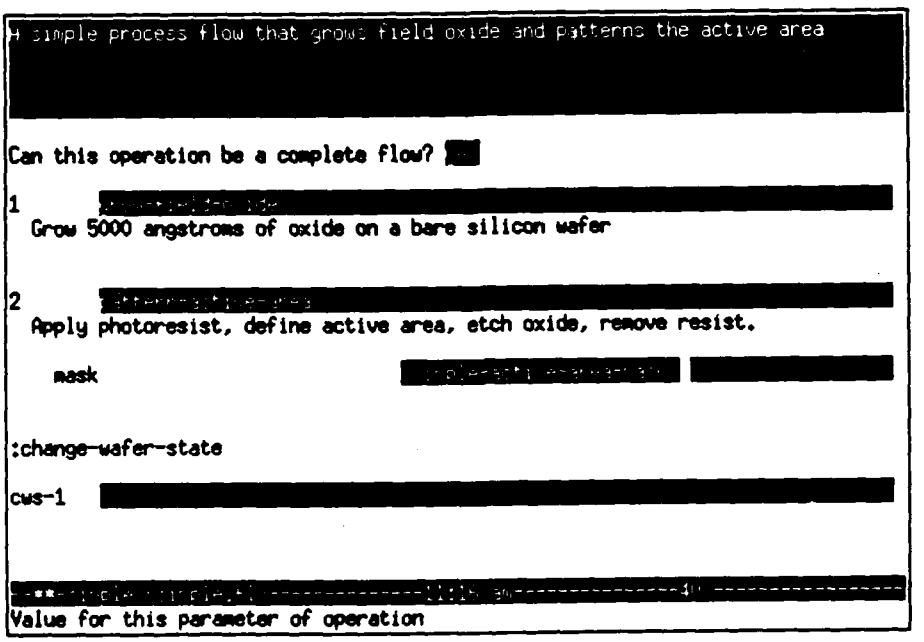

Figure A-5: Completed definition of *simple*

edly to define the lower levels of the flow. *grow-field-oxide* can be defined in terms of operations that perform an RCA clean, grow the oxide, and inspect the thickness of the oxide. Similarly *pattern-active-area* can be defined in terms of operations which perform the photomask step, etch the oxide, and strip the resist. All of the lower level operations should already be defined in some other process flow, such as the CMOS Baseline flow. Operation references can also be inserted within the slots of an operation with multiple slots using the techniques described above. Simply move the cursor to a field within the desired slot before inserting new fields or entering names of operations to reference.

To exit *PFLE*, press ctrl-X ctrl-C until the top-level view reappears. Press ctrl-X ctrl-C one more time to exit the top-level view. *PFLE* will ask whether the updated flow should be written. Answer "y" to write the flow – any other response will discard any changes made to the flow.

Appendix B

Users' Guide to Fabform

User Guide to the *Fabform* User Interface

Rajeev Jayavant

Revised August 3, 1988

Fabform **is a generalized form editor which serves (or will serve) as the user interface for several utilities in the cafe system. This document describes the basic operation of *Fabform* from the user's standpoint. A description of using *Fabform* from a programmer's standpoint will be available in a separate document. All bug reports should be sent to "fabform@caf.mit.edu".**

B.1 Introduction

Fabform is a generalized form editor which serves as the user interface for a number of utilities in the cafe system. Thus most users will never invoke *Fabform* directly, but rather will use *Fabform* as a means of communicating with some other program [1].

The basic concept of a form editor is quite simple. The user is presented with a form, or template, containing a number of blank fields which he may then fill in. *Fabform* displays a screenfull of information and allows the user to move to the various fields in the form. The fields are displayed in either reverse video or underline mode depending on the abilities of the terminal.

The value of a field may be entered or changed by the user only in accordance with a set of rules specified for the form. For example, a field in a form may only be allowed to contain floating point numbers. Another field may be constrained to contain only a member of some specified set (eg. "angstroms" or "nanometers"), and other fields may be specified as "read-only", allowing no modifications whatsoever. *Fabform* will enforce such restrictions, displaying messages if the rules are violated.

B.2 Basic Operation

Emacs users will notice that *Fabform* is similar to *Emacs* in terms of the screen display and the functions bound to most keys. The arrow keys are also supported for cursor movement on VT100, VT200, VT52, HEATH, and related terminals. The exact operation of *Fabform* will vary from application to application because much of the operation depends on the particular form being edited. Nevertheless, this document should provide enough information to use *Fabform* without too much trouble.

B.2.1 Screen Layout

Fabform will display as much of a form as will fit on a screen, allowing the user to scroll vertically through the form if necessary. The last two lines on the screen are used to display status and help information. The second last line is the status line and provides the following information:

modified flag The first few characters on the status line indicate whether the form has been altered since it was last saved. A "**" indicates that changes were made, "--" indicates no changes, and "%%" indicates that the entire form is read-only so changes couldn't have been made. These indicators are just like those used by *Emacs* .

[1] Such as the fabrication interpreter

information area Just to the right of the modified flag is an area that can be used to display any short message chosen by the application using *Fabform*.

time The current time (in 12 hour mode) is displayed in the center of the status line and is updated once a minute.

relative position The right half of the line contains a small indicator of the relative position of the displayed screen to the top of the form. The position is specified as in *Emacs* and may be "Top", "Bot", "All", or a percentage representing the position of the first line displayed on the screen relative to the first line of the form.

The last line on the screen normally displays some help information relating to the field that is currently chosen by the cursor. If an error occurs, the last line of the screen is used to display the error message until the another key is pressed.

B.2.2 Command Summary

More detailed descriptions of some of the commands can be found in later sections of this document. The basic commands are:

ctrl-A if the field can only contain a set of restricted choices, advance to the next valid choice.

ctrl-B move back one field

ctrl-F move forward one field

ctrl-G abort command (really only useful in extended commands)

ctrl-L
- Move currently selected field to the center of the screen.
- If the field is already at the center, refresh the display.

ctrl-M or **RETURN**
- Enter default value if the field has a default.
- Enter current time and date if the field is a blank date field
- Otherwise just advance to next (preferably editable) field.

ctrl-N move down to field on next line

ctrl-P move up to field on previous line

ctrl-Q if the field can only contain a set of restricted choices, reverse to the previous valid choice

ctrl-R scroll screen down one line

ctrl-S scroll screen up one line

ctrl-U enter a repeat count for any cursor movement command

ctrl-V scroll screen forwards by 3/4 of the screen height

ctrl-W delete field

ctrl-X introduce extended commands

ctrl-Y restore field to the value it had before it was visited

ctrl-Z suspend the editor and return to the shell. The editor can be restarted using the "fg" command

ESC introduce more extended commands

DELETE Delete backwards one character.

ctrl-X extended commands

ctrl-G abort command

ctrl-X ctrl-C exit editor. If the form has been modified, the user is asked whether the changes should be saved. If all "required" fields have been filled in, the user is asked whether he has completed all work on the form. If all work is not complete, the editing can be resumed from the point the editor is exited.

Some applications use *Fabform* in *impatient mode*, in which case the user will not be asked whether he wants to exit or save changes. All changes will automatically be saved and *Fabform* will exit.

ctrl-X ctrl-S save the state of the edit session

ctrl-X ctrl-T like ctrl-X ctrl-C but allows the user to declare that he is through editing a form even if some normally required fields are left blank. An edit session terminated via ctrl-X ctrl-T normally will not be continuable whereas one terminated by ctrl-X ctrl-C will be.

ctrl-X ctrl-W write a printable description of the current state of the form to a file. *Fabform* will prompt for the filename to use. Full pathnames must be specified if the file is to reside in any directory other than the current working directory or one of its decendants.[2] The file can later be printed using lpr or

[2]ie. Filenames should not begin with a ~ unless the actual name of the file (as displayed by ls) does.

displayed using **more** or **ul**. ctrl-X ctrl-W is only useful for producing some printable output from *Fabform* – ctrl-X ctrl-S must still be used to save the state of the edit session.

ctrl-X ctrl-Z suspend the editor and return to the shell. The editor can be restarted using the "fg" command.

ctrl-X ? display help information

ESC extended commands

ctrl-G abort command

ESC < move to beginning of form

ESC > move to end of form

ESC **V** scroll screen backwards by 3/4 of the screen height

ESC **X** execute a shell command. The user is prompted for the command to be executed.

ESC DELETE delete backwards until a whitespace is found.

ESC ? display help information

B.3 Cursor Movement

The cursor can be moved from field to field using either the arrow keys or ctrl-B, ctrl-F, ctrl-N, and ctrl-P. The ctrl-N and ctrl-P commands attempt to place the cursor on a field directly above or below the current field. If a suitable field cannot be found, the next-closest field is chosen.

The ctrl-V, ESC V, ctrl-R, and ctrl-S commands attempt to keep the cursor on the current field while scrolling through the form. If the current field is moved off the screen, an attempt is made to move to a field on the new screen closest to the previously chosen field.

The ESC < and ESC > commands move to the first and last fields in the form, respectively.

If there is no visitable field on a screen, the cursor will move to the lower right corner of the display, and all editing operations will be disabled until a field is brought back on the screen using one of the cursor movement commands or the ctrl-L command.

The [RETURN] key can also be used to advance from field to field. The direction of the advance is dependent upon the particular application – some applications prefer advancing towards the right to the next editable field while others advance downwards to the next visitable field. Care must be taken in using the [RETURN] key to advance the cursor since it will also fill in default values of fields (refer to upcoming sections on default and date fields).

B.4 Editing Field Entries

There are only four commands available for editing the value entered in a field. [DELETE] deletes the last character in the field while [ESC] [DELETE] deletes characters backwards from the end of the field until a space is found. The entire field can be deleted using ctrl-W, and ctrl-Y can be used to return a field to the value it had before the cursor was moved onto it.

B.5 Validation of Field Entries

Fabform recognizes a number of field types and restricts a user's input to conform to the requirements of the field. Thus, the actions *Fabform* takes while the user is entering a value into a field depends upon the characteristics of the field.

B.5.1 Unrestricted Fields

Unrestricted fields may contain any printable character (and spaces). Of course, the length of the field is also a limiting factor.

B.5.2 Integer Fields

An integer field can be composed of any string of digits with an optional preceding "-". Attempting to enter anything else will result in an error message and the erroneous input will be disregarded.

B.5.3 Floating Point Fields

A floating point field can be composed of digits, ".", "+", "-", and "e" or "E". If any other character is typed, an error message will be displayed. An error will also occur if the entry is not a properly formatted floating point number (optionally in scientific notation) and an attempt is made to either move to another field or save the field. The entered value is also checked for overflow.

B.5.4 Lisp Expression Fields

Basically any valid printable representation of a lisp object (symbol, number, list, string, etc.) may be entered. *Fabform* will flash the matching open parenthesis when a close parenthesis is entered.

B.5.5 Oneof Fields

Oneof fields can only contain one out of a specified set of choices. A user's input into a oneof field is checked against all possible choices as the entry is made. If *Fabform* detects that the user is not entering a valid choice, an error message is displayed. If, on the other hand, the user enters enough characters to uniquely specify a valid choice, the remainder of the choice is automatically entered. *Fabform* will also attempt to complete partial matches of the user's input to the available choices, thereby minimizing the amount of typing required. Case conversion is performed on the user's input, if necessary, to match the valid choices.

If the user does not know what the possible choices are, or if he just wants to advance through them one by one, the ctrl-A and ctrl-Q keys may be used. If the choice has been partially, but not uniquely, specified, pressing ctrl-A or ctrl-Q will display the first choice that matches the partial specification.

B.5.6 Default Values

All of the field types described above may have an associated default value. The help line at the bottom of the display will contain "(default available)" whenever a field with a default value is selected. The default value, however, is not used unless it is explicitly selected by pressing RETURN or ctrl-M when the field is selected. If a different value is entered for the field, the default is lost.

B.5.7 Date Fields

Date fields are similar to Default fields except that the current date/time is the default. Simply press RETURN to enter the current date into the field. Only the month and day need to be entered if the current date is not wanted. The defaults for the remaining (unspecified) parts of the date are the current year for year and zero for hours, minutes, and seconds.

B.5.8 Read-only Fields

Read-only fields, as their name implies, cannot be modified. When the cursor is on a read-only field, the help line at the bottom of the screen will display "(read only)". *Fabform* will display an error message if you attempt to modify a read-only field.

The read-only status of a field can change over time. A form may contain fields that change to read-only status after a save operation [3].

B.6 Saving Entered Field Values

The field values entered by the user are not saved until an explicit save is performed either via ctrl-X ctrl-S or during exit from *Fabform*. If the program dies for any reason (eg. due to a machine crash), any changes made since the last save will be lost. Thus it is advantageous to save often.

On the other hand, some forms contain fields that become read-only after a save. Thus extra care should be taken to ensure that the entries in such fields are correct before performing a save.

B.7 Exiting *Fabform*

B.7.1 Temporary Suspension

Fabform may be temporarily suspended using either the ctrl-Z or ctrl-X ctrl-Z commands. *Fabform* can restarted later using the "fg" command from the shell or by selecting the appropriate choice in the "Tasks" submenu in the wand.

Some applications may disable the the suspension mechanism, in which case ctrl-Z and ctrl-X ctrl-Z will only generate a warning message. In the absence of the suspension mechanism, the only way to exit *Fabform* is via one of the quit commands.

The most common reason for using a suspension mechanism is to temporarily stop a process, execute some other commands in the shell, and then resume the original process. If the suspension mechanism is disabled, shell commands can still be run using the ESC X command.

B.7.2 Permanent Exit

There are two commands for permanently exiting *Fabform*: ctrl-X ctrl-C and ctrl-X ctrl-T. In many applications, there is no difference between the two commands, with ctrl-X ctrl-T defaulting to ctrl-X ctrl-C. In most cases, ctrl-X ctrl-C should be used to exit *Fabform* – ctrl-X ctrl-T is normally used only when the user has finished all editing [4] of a form but wishes to leave some fields blank. The properties of the two commands can be summarized as follows:

---

[3] eg. to prevent history from being modified

[4] ie. he never wants to edit that particular form again

ctrl-X ctrl-C This can be thought of as a long term suspension. If any changes have been made, the user is asked if the changes should be saved before exiting (allowing a quick way to abandon any changes made to the form). *Fabform* terminates after confirming the user's intentions [5] , but indicates to its associated application that editing of the current form is to be resumed at some later time. If, however, all fields in the form have been filled in, ctrl-X ctrl-C defaults to ctrl-X ctrl-T.

ctrl-X ctrl-T *Fabform* saves all changes and exits after confirming the user's intentions.[6] The associated application is *told* that the user has completed all editing on the current form and does not wish to return to it at a later time. If some fields have not been filled in, the user is warned of their existence and asked for a confirmation of the command.

[5] except when an application uses *Fabform* in *impatient mode*. All changes will automatically be saved and *Fabform* will exit without confirming intentions.

[6] except when *impatient mode* is in effect

Appendix C

Programmers' Guide to Fabform

Programmer's Guide to the *Fabform* User Interface

Rajeev Jayavant

Revised August 3, 1988

Fabform is a generalized form editor which serves as the user interface for several utilities in the cafe system. This document describes the basic operation of *Fabform* from the programmer's standpoint, providing the information necessary to use *Fabform* as the user interface for a given application. The User's Guide to *Fabform* should be consulted for further information. Questions and bug reports should be sent to "fabform@caf.mit.edu".

C.1 Introduction

Fabform is a generalized form editor designed to be used as the user interface for a variety of programs. It is quite flexible in terms of the types of forms that can be handled, though the attempt to keep *Fabform* as generalized as possible imposes a few limitations. Terminal independence is an important feature of *Fabform*, though performance degrades rapidly when running on terminals with fewer capabilities than a vt52. This document describes the capabilities and deficiencies of the current implementation of *Fabform*.

Software updates and bug fixes are made periodically. Every attempt will be made to keep new releases of *Fabform* completely compatible with applications designed around older versions. Please send a bug report if you suspect that a problem is introduced in a new release that makes it incompatible with previous versions. If you feel that your application (and hopefully others) could benefit significantly by an addition of features to *Fabform*, please send mail to "fabform@caf.mit.edu" and describe your situation. Perhaps the feature can be added if the modification is feasible.

C.2 Interaction with *Fabform*

Fabform is designed to run as a separate process and communicates with the associated application program via two files: the template file and the parameter file. The template file describes the layout of the form and the types of data that can be entered in the various fields. The parameter file specifies the contents of the various fields defined in the template file. *Fabform* takes a template and/or parameter file as input and produces a parameter file as output. Both input and output parameter files may be read from/written to pipes if desired. The `examples` directory under the *Fabform* source code directory contains some sample application programs that interact with *Fabform*.

Once the application program invokes *Fabform*, it can no longer communicate with *Fabform*. All interaction between the two processes is done only through the files described above, and the input template and parameter files are read only when *Fabform* is invoked. The output parameter file, however, can be read at any time, so the application can keep track of changes to the output file as they happen[1]. If the output parameter file is actually an output pipe, the process reading the

[1] The output parameter file is first written as soon as *Fabform* finishes reading the input parameter file and is updated whenever the user executes a *save* command

information sent down the pipe can monitor data as it is saved by the user.

The *format* of the output parameter file is identical to that of the input parameter file (described later). The actual ordering of information within the output parameter file can vary greatly from that in the input parameter file, thus programs reading the parameter files should not make any assumptions about the ordering of information. The only ordering that is guaranteed in the output parameter file, in addition to the ordering imposed by the restrictions on parameter files, is that multiple instances of an operation appear in the same order as they appear in the form being edited. If the output parameter file is actually a pipe, the EOF character (control-D or '\004') is used to indicate the end of a complete set of data[2].

Whitespace is added to the output parameter file in an attempt to make it more readable (presumably for debugging purposes). No assumptions should be made about the existence of whitespace since the specifications of the parameter file format state that whitespace should be ignored.

The exit status of *Fabform* determines whether the user has completed all editing of the form.[3] Exit status 0 indicates that all editing is complete, status 1 indicates the user wishes to return to the form at some later time, and a negative status indicates a fatal error occurred[4].

The template files are intended to be static in the sense that under most circumstances they would be created once and used repeatedly. Parameter files, on the other hand, are normally created dynamically as they are needed and deleted when they are no longer needed.

---

[2]The output parameter file is updated whenever the user executes a *save* command. If the output is to a pipe, there is no way to *rewind* the file to the beginning, thus the EOF character is used to indicate the end of a block of data that would normally correspond to a complete output parameter file.

[3]ie. Does the user want to resume editing the current at some later time or has he/she made all the modifications that are ever to be made to the form? A form is said to be complete when the user has entered values in all fields that are required to be filled in, or if the user claims that he has completed filling in the entire form.

[4]Fatal errors can be caused by errors in the template or parameter files, by failure to open the specified files, refusal of the system to grant a memory allocation request, or a fatal error in *Fabform* itself.

C.3 Template File Format

The template file format resembles LaTeXfiles in that all command directives begin with a "\" and any amount of blankspace[5] is treated as a single space. A \ may be included in the form by specifying \\ in the template file.

C.3.1 Positioning Commands

\hspace[#chars] insert horizontal blankspace of #chars

\hpos[column#] move to horizontal position given by column#. First column is 1

\nl begin new line

\vspace[#lines] insert #lines blank lines. \vspace[0] is equivalent to \nl

Any blankspace in the file before or after \hpos or \hspace is ignored. Blankspace is also ignored if it occurs at the beginning of a line.

C.3.2 Field Definition Commands

Field definitions consist of a field width, a field name[6], and some help information describing the purpose of the field. Some fields implicitly define the width while others do not have the help information because users are not allowed to move the cursor onto those fields. The help information is normally displayed on the last line of the screen when the user places the cursor on a field. The restriction on field width is determined at compile time, though the typical maximim field width is over 2000 characters. Thus there is no problem having fields that span line breaks.

The field declaration describes the type of data that may be entered into a field. The parameter file may place further restrictions on the type of data that may be entered.

The field definition commands are:

\global[width][name] insert global parameter for specified width. A global parameter is one that can be accessed from within any operation block (see next

[5] Blankspace is defined as any combination of spaces, tabs, and newlines.

[6] Field names will be also be referred to as *tags* or *parameters* in later sections of this document.

section) but cannot be entered by the user. If the value of a global is not specified in the parameter file, the field is left blank. There are a few globals that are defined by *Fabform* if they are not specified in the parameter file.

user the login name of the user running *Fabform*

now the time at which *Fabform* was invoked. The time is in the format "mm/dd/yy hh:mm:ss"

editor-name some default string that is printed on the status line

help-file the file containing a summary of *Fabform* commands

integer[**width**][**name**][**help**] allow only the digits 0 through 9 with an optional preceding minus sign

float[**width**][**name**][**help**] allow only valid floating point numbers, including scientific notation

string[**width**][**name**][**help**] allow any arbitrary string of printable characters

comment[**width**][**name**][**help**] just like \string except that the field does not become read-only after a save. This field type is equivalent to \string*- or \string-* (see below) and exists only for backward compatibility.

readonly[**width**][**name**][**help**] just like \string but always read-only (but visitable, unlike \hidden)

lispexp[**width**][**name**][**help**] Allow only valid "lisp" expressions. Tests for matching double-quotes and parentheses. The matching open parenthesis is flashed whenever a close parenthesis is entered.

hidden[**width**][**name**] If the value is not specified in the parameter file, the field is left blank. The user cannot move the cursor onto a hidden field. Very similar to a global except that the name lives in the local namespace of the operation (see next section).

date[**name**][**help**] user can enter the current time into the field by pressing RETURN. The date is in the format "mm/dd/yy hh:mm:ss". If the current time is not desired, the user may enter any valid date and time. Only month and day need to be specified with the year defaulting to the current year and hours, minutes, and seconds defaulting to zero.

day[**name**][**help**] Just like \date except that the field does not include the time of day. The day is in the format "mm:dd:yy". The strange naming of these two field types is the result of preserving backward compatibility.

\global, \hidden, and \readonly fields are always read-only (and only \readonly is visitable) whereas \comment fields are always writable. All other field types are write-once; the fields are writable until the user enters a value and *saves* the state

of the form. Appending a "*" to any field type forces the field to remain writeable after a save. Appending a "-" to any field type signifies that filling in the field is optional, thus the field is considered to be filled when *Fabform* determines whether the user has completed filling in the form.

C.3.3 Operation Block Delimiters

All tags other than the globals live in the local namespace of the operation block they are defined in. Thus it is possible to reuse tag names as long as they occur in different operation blocks. There may be multiple instances of the same operation block, but operation blocks may not be nested. Names of tags, globals, and operation blocks (described below) are case-insensitive.

An operation block must be started before the first non-global field can be defined, and an operation block must end before the next one can begin.

The block delimiting commands are:

\begin[operation_name] defines the start of a new operation block.

\end[operation_name] defines the end of an operation block. The operation_name specified must match the name specified in the \begin command.

C.4 Parameter File Format

The parameter files use a lisp-like syntax to delimit different operation instances that may exist within a single parameter file. The structure of the files is quite flexible. Whitespace (spaces, tabs, and newlines) is ignored except when it occurs within double-quotes ("). Since double-quotes are used to delimit strings, they must be preceded by a \ if they are used within a string. Similarly, a \ is represented as \\ within a quoted string. Information within a parameter file is case-insensitive (except for quoted strings) and will be represented in lower-case in the output parameter file. Additional restrictions are described at the end of this section.

The following three strings are keywords and should be avoided as tag names to prevent any possible confusion. None of these directives may be nested within each other.

template specifies a template file to use to extend the current form

operation defines a block of parameters for an operation instance. Details on syntax follow. Parameters declared/defined within a particular operation instance remain local to that instance, allowing multiple instances of the same operation.

define-oneof-class define a class of objects that can later be specified using the defined class name (see **oneof-class**).

The following directives may be specified only within parameter definitions within an *operation* directive. These directives are not guaranteed to work if they are nested, but they may be used in conjunction with each other as described later in this section.

default defines a default value for a parameter that does not have a value assigned to it. The default value does not become the value of the parameter until the user visits the field and presses RETURN.

initial-value defines an initial value for a parameter and allows the user to modify it. This differs from simply giving a value to a field in that fields that become read-only may be given an initial value that can be modified.

private assigns a value but does not allow the user to tell whether the value is really a parameter or something hardwired into the template. Good for hiding things from users who are better off not knowing the truth.

oneof restrict the parameter to take on one of the specified values

oneof-class like **oneof** but specifies the name of a class defined via **define-oneof-class** rather than a list of choices

exec-function function to execute when the value of the field is changed. Refer to the Addendum to the Programmer's Guide to Fabform for more details

In addition to the above directives, *Fabform* supports a class of values called globals that may be accessed from any part of a template file. Globals are defined at the start of a parameter file using the syntax:

```
(global-parameter "value")
```

Fabform also provides default values for four globals:

user the login id of the user

now the time at which *Fabform* began execution

editor-name the message displayed in the status line. default is set at compile time at the whim of the programmer.

help-file the name of the file to display when the user asks for help. Normally this is a file containing a summary of *Fabform* commands.

A parameter file may define globals that do not exist anywhere in a template (allowing information to be transferred from the input parameter file to the output parameter file), but any local field definitions (ie. within operations) must correspond to fields defined in a template file.

The *template* directive allows many separate template files to be appended together to create a single form. Each *template* directive causes a separate template file to be appended in the creation of the form. The syntax is:

```
(template "template-file-spec")
```

The *operation* directive takes the form

```
(operation operation-name
        <parameter value assignments>
        ...
)
```

The *define-oneof-class* directive is useful if many fields are to be restricted to the same set of values. Rather than specifying the restriction list explicitly within each parameter description via a oneof declaration (see below), the restriction list can be defined once using the *define-oneof-class* using the form

```
(define-oneof-class class-name choice_1 choice_2 ...)
```

The defined class can then be referenced via the *oneof-class* declaration within a parameter definition. Class names defined using *define-oneof-class* live in their own namespace, but this namespace is common to all operation instances. Thus class definitions are global to all procedures and a class may not be redefined. A class name may, however, be the same as that of a global or a tag within an operation.

Parameter values are assigned using one of the following forms:

```
(parameter-name value)
```

```
        (parameter-name (default value))
        (parameter-name (initial-value value))
        (parameter-name (private value))
        (parameter-name (oneof choice_1 choice_2 ... choice_n))
(parameter-name (oneof-class class-name))
```

where value may be any arbitrary string. Whitespace will be deleted unless value is enclosed in double quotes. The double quote character therefore may not be included as part of any value. All values will be truncated to the field width specified in the template file, if required. The *oneof* (or *oneof-class* and *default* directives may be used in conjunction, e.g.

```
        (parameter-name (default-value)
                        (oneof choice_1 choice_2 ... choice_n))
```

In the case of fields that do not become read-only after a save, it is even sensible to specify a value in addition to a *oneof* declaration, eg.

```
        (parameter-name value
                        (oneof choice_1 choice_2 ... choice_n))
```

Parameter names may also be specified hierarchically, eg.

```
        (measurements
            (right
                (value 2378)
                (units "A"))
           (left
                (value 2412)
                (units "A")))
```

Internally the parameter names are treated as "measurement.right.value", "measurement.right.units", "measurement.left.value", and "measurement.left.units". These expanded names are the ones that must be used within the template file.

A parameter file consists of any number of *global definitions* followed by any number of *operation*, *template*, and *define-opset-class* directives. The only restrictions are that all *global definitions* must appear before any *template* directives, the *template*

directive corresponding to an *operation* directive must occur **before** the *operation* directive, and a *define-opset-class* directive must occur **before** the class-name is referenced in a *oneof-class* declaration. The number of operation and template directives is largely determined by the capability of the VM system and the patience of the user for going through long forms.

The format of the parameter can be parsed easily by a Lisp application, but C applications may find the job more difficult. To ease the burden for C programmers, a collection of routines for parsing parameter files is provided in the `utilities` directory under the *Fabform* source directory, along with the accompanying documentation.

C.5 Invoking *Fabform*

The *Fabform* command line syntax is as follows:

```
/usr/cafe/lib/fabform [-p input_parameter_file]
                      [-o output_parameter_file]
                      [-d template_directory] [-t template_file]
                      [-f output_form_file] [-r] [-v] [-q] [-n]
      [-s field_name:op_name:op_instance]
                      [-H help-file] [-N editor-name]
                      [-z] [-i] [-e] [-D]
```

Any or all of the flags may be specified. Either an input parameter file or a template file must be specified in all cases. The command line flags are interpreted as follows:

-p Use the next argument as the name of the input parameter file. If a template file is also specified with the -t flag, the template file will be read first.

-o Use the following argument as the output parameter file. The output file may be the same as the input file without any chance of data loss. The output file is written whenever the user performs a save operation and is guaranteed valid unless there is a shortage of disk space or the system crashes during a write.

If the output filename begins with a "|", pipe the output to that process instead of writing it to a file. For example,

```
fabform -p infile -o "|foo arg1 arg2"
```

will pipe the information that would be sent to the output parameter file into the process "foo arg1 arg2". An EOF character (control-D, or '\004') will be sent down the pipe to indicate the end of a save operation (since the user may repeatedly save his data).

-d The next argument will be used as the directory in which to look for template files. A trailing / will be added if necessary.

-t Use the following argument as the name of the template file to read. If an parameter file is also specified, the template file will be read first. The -t option is useful for generating skeleton parameter files from a template file (eg. "fabform -t templatefile -o outputparamfile -f /dev/null") or for using different templates with a static parameter file.

-f Use the next argument as the filename to store a printable representation of the form into. The file can then be printed using *lpr* or displayed on the screen using *ul* or *more*. *Fabform* does not start up interactively and simply exits after creating the form file.

-n The default movement upon pressing RETURN is changed to vertical instead of horizontal

-q Do not attempt to report whether all editing is complete. Normally *Fabform* will attempt to determine whether all editing is complete on a given form and return an appropriate exit status, sometimes by querying the user. In situations in which the notion of completeness does not apply, the -q option should be used to prevent the user from being unnecessarily harassed. If -q is specified, ctrl-X ctrl-T behaves like ctrl-X ctrl-C, and the exit status of *Fabform* is meaningless if it is non-negative.

-r Read-only mode. Allow the user to browse through the form without changing anything. No output parameter file is created even if -o is specified.

-H specify the file to print as the help file. Can be overridden by a specification in the parameter file.

-N specify the text to use for the "editor name". Can be overridden in the parameter file.

-z Disable suspension via ctrl-Z or ctrl-X ctrl-Z. Useful if you don't want the user to suspend the application.

-D Display messages for non-fatal errors in parameter or template files. If any such errors are detected, *Fabform* terminates after reading the entire parameter or template file. If the -D flag is not specified, non-fatal errors are not reported and operation of continues as usual. The non-fatal errors are:

- Definition of parameters not used in the corresponding template file. The definition is ignored.

- Specifying an undefined *class* in a **oneof-class** declaration. The **oneof-class** declaration is ignored, and no additional restrictions are put on the field.
- Multiply defining a *class* in **define-oneof-class** declarations. Only the first definition is stored.
- A reverse \hpos while reading a template file

-i Use *impatient mode*. Do not verify user's intentions for ctrl-X ctrl-C, ctrl-X ctrl-S, or ctrl-X ctrl-T. A save is automatically performed before an unconditional exit.

-e Initially position cursor on last editable field in the form instead of the first.

-s Use the next argument to determine where to position the cursor at the start of the edit session. The cursor will start out on the field named `field_name` in the `op_instance`th instance of the operation named `op_name`.

input pipe The input parameter file can be read from an input pipe rather than a file. The parameter file must be *piped into Fabform* using a mechanism similar to the **popen()** function or the piping mechanism of a shell. The **-p** option overrides use of an input pipe to read the input parameter file.

output pipe The output parameter file can be written to an output pipe rather than a file. The output of *Fabform* can be *piped into* another process using a mechanism similar to the **popen()** function or the piping mechanism of a shell. *Fabform* can use this mechanism to pipe the output parameter file to an existing process (eg. its parent). The **-o** option will override the use of an output pipe[7]

[7] An output pipe created using the **-o** option first creates a new process into which the output parameter file will be piped. Using an output pipe without the **-o** option allows communication with an existing process.

Appendix D

The Procedural Interface to Fabform

The initial version of *Fabform* ran as a separate process and only communicated with an application through files. The application created files representing the initial state of the form and handed them to *Fabform*. When *Fabform* exited, the application would read the files to determine the final state of the form.

While this type of interaction was adequate for certain applications, e.g. the machine reservation program, there was a need for a mechanism that allowed finer-grained interaction between an application and *Fabform*. Thus a procedural interface to *Fabform* was created, allowing *Fabform* to be embedded within an application and call functions within the application whenever the user performed actions of interest to the application. The *Addendum to the Programmers' Manual to Fabform* documents this procedural interface.

Addendum to Programmer's Guide to the *Fabform* User Interface

Rajeev Jayavant

Revised July 1, 1988

Fabform is a generalized form editor which serves as the user interface for several utilities in the cafe system. This document describes how *Fabform* may be incorporated as a subroutine in a user program. *Fabform* now includes support for applications written in Lisp as well as in C.

The Programmer's Guide to *Fabform* contains additional information that is required in order to use *Fabform* within an application. The User's Guide to *Fabform* should be consulted for further information. Questions and bug reports should be sent to "fabform@caf.mit.edu".

D.1 Introduction

One major disadvantage of running *Fabform* as a separate process in the minimal interaction between the application and *Fabform* while the user is editing the form. By incorporating *Fabform* into the application itself, it becomes possible to associate a function [1] with each field. The function will be called whenever the value of that particular field changes [2]

The called function may modify the behavior of *Fabform* upon return in any of the following ways:

- change the value of the current field
- display a message in the status line
- make the current field writable again (if it was made read-only by a save just prior to calling the function)
- save the output parameter file
- exit *Fabform*

In addition, the function may perform any actions it wants to, including writing to the user's terminal or modifying the signals. To avoid confusing *Fabform* upon return, mechanisms are provided to instruct *Fabform* to reset its signals and refresh the screen.

A final bonus is that *Fabform* may be called recursively with minimal overhead since there is no need to start a new process. Of course, if very large forms are used, there is a chance of running out of memory space.

The only real disadvantage to including *Fabform* as a subroutine in an application is that the application must be relinked to incorporate the latest bug fixes and enhancements made to *Fabform*. [3] When *Fabform* is run as a separate process, the most recent version is automatically available.

---

[1] The functions may be written in C or Lisp, and all of the functions do *not* have to be written in the same language.

[2] The function declaration specifies whether to call the function when a new value is "entered" into the field or to call it as individual characters are entered into the field.

[3] A Lisp application may choose to load the *Fabform* package at runtime, thereby automatically obtaining all updates.

D.2 The *Fabform* Interface

There are two routines which may be executed to start a *Fabform* edit session, `fabform()` and `fabforml()`. The routines differ only in the format in which the arguments are specified. Lisp applications must use the macro `fabform:exec-fabform` which takes very similar arguments.

```
int fabform(function_defs,args)
    struct fabform_functions *function_defs;
    char *args[];

int fabforml(function_defs,arg1,arg2,...,argn,(char *)0)
    struct fabform_functions *function_defs;
    char *arg1,*arg2,...,*argn;

(fabform:exec-fabform function-definition-list arg1 ... argn)
```

D.2.1 Function Definitions

The `function_defs` argument specifies a list of functions, names, and instructions for running those functions. The `function-definition-list` in the Lisp macro performs the same role. The actual assignment of a function to a field in the form is done via the input parameter file using the `exec-function` construct. For example, to attach the function named "twiddle" to a field named "knob", the parameter file entry would look like:

```
(knob (exec-function "twiddle"))
```

Of course, other assignments can simultaneously be made to "knob" (e.g. oneof, oneof-class, default, private, or even a value). The name given in an `exec-function` construct must match a name in the `function_defs` list passed to *Fabform*, otherwise no function assignment will be made to the field.

The format of the `function_defs` entries is as follows:

```
struct fabform_functions {
   struct fabform_retstat (*function)();
   char *name;
```

```
    int action;
}
```

The fields in the structure have the following meanings:

function a pointer to the function to be executed. The calling interface of these functions will be discussed later in this document.

name the name that this entry will be referenced by in an `exec-function` construct in a parameter file

action the actions that will be taken *before* the function is called. The function may specify additional actions to be taken upon returning (discussed later). The actions field should be constructed by ORing (use |) any of the following values together.

FABFORM_SET_SIGNALS resets SIGINT, SIGQUIT, SIGTSTP, SIGBUS, SIGSEGV, and SIGALRM to their original handlers. The signals will automatically be restored to *Fabform*'s preferred settings upon return. Any function that wants to take control for an extended period (e.g. to interact directly with the user or start up another process) should use this feature. See section on Signal Handling for more details.

FABFORM_SAVE save the state of the form into the output parameter file before calling the function. Allows the function to look at the values of other fields in the form. Be careful with this one since some fields can become read-only after a save.

FABFORM_EXIT exit *Fabform before* calling the function. This can be useful if the function always exits *Fabform* and has the advantage that memory allocated by *Fabform* is released *before* the function is called.

FABFORM_NO_REFRESH do not save the tty state and keep the cursor in the current position. This value should only be included if the function is not going to affect the screen in any way whatsoever. If FABFORM_NO_REFRESH is missing, the tty state is saved and the cursor is moved to the lower left corner. A screen refresh is performed on return only if the return status requests one.

FABFORM_CLEAR Clear the screen before calling the function. A refresh will be performed upon return even if one is not requested.

FABFORM_WRITABLE keep this field writable even if it is made read-only by a save

FABFORM_REEDIT keep cursor on present field after return

FABFORM_ANY_KEY call the function any time the user presses a key that leaves a valid value in the field[4]. Normally the function is called only when the user "enters" a value in a field (e.g. by pressing RETURN or moving to another field).

FABFORM_EXEC execute function. This one will automatically be set if it is missing. Kind of useful to have something to assign, though, if none of the other options are desired.

The final entry in the `function_defs` list must have the name field set to `(char *)0`. If no function definitions are to be made, `function_defs` may be specified as `(struct fabform_functions *)0`.

The Lisp `function-definition-list` is used to express the same information as `function_defs` is in C. The `function-definition-list` is a list of function definitions of the form:

```
(function-pointer function-name action)
```

The fields of the function definition (which is also a list) are very similar to their C counterparts.

function-pointer A Lisp integer that may be passed to C as a pointer to an executable C function. A suitable pointer to a Lisp function is obtained by using the macros `fabform:make-fabform-function` or `fabform:defun-fabform-function` to define a Lisp function (discussed in the next section).

function-name The name the function will be referenced by in the `exec-function` entry of a parameter file.

action A list composed of the following keywords whose actions correspond to those of their C counterparts.

- :clear
- :set–signals
- :save
- :exit
- :no–refresh
- :writable

[4]The function is not called when the *new* value is obtained by deleting characters. This is currently considered a *feature* and may not be present in future implementations.

- :reedit
- :any–key
- :exec
- :no–exec

The action *list* may be *nil*.

If the first entry in the `function_defs` list is named "fabform_init", the function declared in that entry will be called before any editing is begun but after all field definitions and values have been read from the specified files. The "fabform_init" function is like any other function associated with a field except that all the parameters it is called with will be passed in as NULL or 0 when it is called as an initialization routine.

D.2.2 *Fabform* Options

The arguments specified in `args` or `arg1` through `argn` are equivalent to the command line options for *Fabform*. If the `fabform()` routine is used, the last entry in `args` should be `(char *)0` to indicate the end of the list.

`fabform1()` provides a simpler interface to use if the number of arguments is known at compile time. There is a limit of about 100 arguments for this interface while the number of entries in `args[]` for `fabform()` is unlimited.

All input to *Fabform* is still in the form of parameter files and template files. Applications making full use of functions associated with fields, however, may find it unnecessary to use an output parameter file. Such applications should simply not specify an output file and save everyone from doing unnecessary work.

D.2.3 Return Value

The value returned by `fabform()` and `fabform1()` is identical the exit status returned by a *Fabform* process, except if a function requests an exit. Briefly, the exit status can be:

<0 fatal error. an error message describing the problem should be generated

0 user exited form and claims that all entries are filled

1 user exited form and wishes to re-edit at a later time

>1 a function associated with a field requested the exit. The actual return value is meaningless to an application.

D.3 The Function Interface

The functions associated with fields must be of the form:

```
struct fabform_retstat
     function_to_exec(field_value,field_name,operation_name,
                      operation_instance,output_file_name)
            char *field_value,*field_name,*operation_name;
            int  operation_instance;
            char *output_file_name;
```

where the operands have the following meanings:

field_value the contents of the field

field_name the name of the field whose contents were changed. Allows the same function to be associated with many different fields. The field_name will always be in lower case.

operation_name the name of the operation in which the field is defined [5] The operation_name will always be in lower case.

operation_instance indicates which instance of the operation the field is defined in [6]. The first instance of an operation is numbered 1.

output_file_name the name of the output parameter file in case the function is interested in the state of globals, etc. Much more useful if the FABFORM_SAVE attribute is set in the `function_defs` entry for the function. The values of other fields and their restriction lists can be examined and modified via a set of functions.

The return value of the function specifies the actions that *Fabform* should take. The `fabform_retstat` structure is defined as:

---

[5] It is possible to use the same field name in different operations (or \begin[]/\end[] blocks).

[6] An operation name does not uniquely identify an operation instance; an operation of a given name can occur any number of times within a form.

```
struct fabform_retstat {
  char *field_value;
  char *message_string;
  int action;
}
```

All of the fields must be assigned using the following guidelines.

field_value The contents of the field can be changed by specifying a pointer to a character string in the `field_value` entry of the return value. The specified string should be in a static location, not in the stack frame of the called function since this stack frame will be destroyed upon return. To leave the contents of the field unchanged, assign the `field_value` entry to `(char *)0`.

message_string *Fabform* will display the string that the `message_string` entry points to. Once again, the string should be in a static location and not in the stack frame of the called function. If the `message_string` entry is assigned to `(char *)0`, no special message will be displayed.

action Specifies what actions *Fabform* should take upon the function's return. If no special action is to be taken, the `action` entry should be set to FABFORM_NO_EXEC. Otherwise it should be set to the logical OR of any of the following values.

FABFORM_SET_SIGNALS restores signals to a state that *Fabform* prefers them to be in. This option should be used anytime a function changes signal handlers, though it is unnecessary if FABFORM_SET_SIGNALS was specified in the `function_defs` entry for the function. When in doubt, specify FABFORM_SET_SIGNALS.

FABFORM_SAVE save the state of the form into the output parameter file. Can be handy if used in conjunction with FABFORM_EXIT. Be careful with this one since some fields can become read-only after a save.

FABFORM_EXIT exit *Fabform*. Allows quick exits without having to enter ctrl-X ctrl-C all the time.

FABFORM_NO_REFRESH do not refresh the screen upon return. Normally the screen is refreshed to ensure integrity of the display, but any function that can guarantee no output to the terminal may set this value to avoid unnecessary refreshes.

FABFORM_WRITABLE keep this field writable even if it is made read-only by a save

FABFORM_REEDIT keep cursor on present field after return

D.3.1 The Lisp Function Interface

Lisp functions associated with fields must be of the form

```
(defun function-to-exec (field-value field-name
operation-name operation-instance
output-file-name)
        (...)
        ...
        (fabform:return-values new-field-value
                               message-string action))
```

The parameters are identical to their C counterparts. Since there is a problem in returning values directly from Lisp to C, the Lisp function must call `fabform:return-values` to return values to *Fabform*.[7] The `new-field-value` and/or `message-string` arguments should be `nil` if the field value and/or message string are not to be modified upon return. The `action` field should be a (possibly null) list of keywords from the following set (:set–signals, :save, :exit, :no–refresh, :writable, :reedit, :any–key).

The Lisp function can be made callable by Fabform by using value returned by

```
(fabform:make-fabform-function function-to-call)
```

in the function definition list passed to `fabform:exec-fabform`. Since `fabform:make-fabform-function` creates a new function and registers it in a table of C-callable functions, it should not be called repeatedly for the same Lisp function to avoid overflowing the table.

Alternately, the macro `defun-fabform-function` can be used. It uses the same format as `defun` but returns a value that may be used in a function definition list passed to `fabform:exec-fabform-function`.

D.4 Signal Handling

Fabform uses its own signal handlers for SIGINT, SIGQUIT, SIGTSTP, SIGALRM, SIGBUS, and SIGSEGV. The SIGTSTP, SIGBUS, and SIGSEGV handlers invoke

[7] The *real* return value of the Lisp function should be the value returned by the call to `fabform:return-values`.

the original handlers after doing their thing.

Whenever *Fabform* sets these signal handlers, it first checks to see whether the presently assigned handler is different from what it wants to use. If the old handler is different, it is saved for later use when *Fabform* resets signals.

Functions associated with fields may modify signal handlers under two conditions:

1. The FABFORM_SET_SIGNALS bit is set in the `action` field of the returned value. This will tell *Fabform* to reinstall its own signal handlers.
2. The function must be aware that any signal handler that it changes becomes the new *saved* handler which will be reinstalled whenever *Fabform* resets the signal handlers.

Fabform will reset signal handlers (ie. reinstall the saved handlers) upon a normal exit. Any functions which do a recursive call to *Fabform* should be sure to set the FABFORM_SET_SIGNALS bit in the `action` field of the return value[8]. Upon abnormal exit from *Fabform*, everything is restored to the state that existed when that invocation of *Fabform* was begun.

D.5 Utility Routines

The following routines may be called from functions associated with fields. They allow simple methods by which data or routines internal to *Fabform* may be accessed while retaining the safety of a level of abstraction.

```
char fb_display_message(message)
        char *message;

(fabform:display-message message)
```

Display the message in the status line on the last line of the screen. An older interface, fb_display_message_prompt() is also available to C routines that leaves

[8]The saved signal handlers will be reinstalled by the recursively-called *Fabform* when it exits. Since there is only one set of saved handlers for *all* inocations of *Fabform*, very bad things will happen if *Fabform* is not told to install its preferred set of handlers. The FABFORM_SET_SIGNALS bit should be set whenever there is any doubt whether *Fabform*'s preferred set of handlers is installed. The worst thing that can happen by setting the bit is having do a little unnecessary computation.

the cursor at the end of the message being displayed rather than returning the cursor back to the field it was on.

```
char fb_query_yes_no(prompt)
        char *prompt;

(fabform:query-yes-no prompt)
```

Display the prompt on the last line of the screen and wait for the user to press "y", "Y", "n", "N", or control-G. Returns either 'y', 'n', or '\007' (or the equivalent characters in Lisp).

```
char fb_query_single_keystroke(prompt,validchars)
        char *prompt,*validchars;


(fabform:query-single-keystroke prompt validchars)
```

Display the prompt on the last line of the screen and wait for the user to press a single keystroke. If `validchars` is non-NULL, wait until user presses a keystroke that is in `validchars`.

The Lisp version returns a character object. `nil` may be specified for `validchars`.

```
int fb_query_string_value(prompt,string,allow_spaces)
        char *prompt,*string;
        int allow_spaces;

(fabform:query-string-value prompt allow-spaces)
```

Displays the specified prompt on the last line of the screen and allows the user to enter characters as long as space permits on the bottom line. If `allow_spaces` is non-zero, the user may enter spaces, otherwise, they are ignored. The user's input is returned in `string`, and the return value is the length of the input. A -1 is returned if the user pressed control-G.

The Lisp version returns the string entered by the user or `nil` if control-G was pressed. `allow-spaces` should be specified as `nil` if spaces are not to be allowed.

```
char * fb_get_field_value(field_name,op_name,op_instance)
        char *field_name,*op_name;
        int op_instance;

(fabform:get-field-value field-name op-name op-instance)
```

Return the value of the field named `field_name` within the `op_instance`th occurance of operation `op_name`. Returns NULL if the field could not be found.

```
int fb_set_field_value(field_name,op_name,op_instance,
                       field_value,refresh)
        char *field_name,*op_name,*field_value;
        int op_instance,refresh;

(fabform:set-field-value field-name op-name op-instance
 field-value refresh)
```

Sets the value of the field specified by `field_name`, `op_name`, and `op_instance` to `field_value`. No validation is done on the value, though truncation is performed if necessary. Read-only and hidden fields can be *modified* using `fb_set_field_value()` if desired. If `refresh` is non-zero, the new value will immediately be displayed on-screen, otherwise it will be updated whenever the screen is refreshed. The return value will be 0 if successful, -1 if the specified operation instance could not be found, and -2 if the field could not be found.

```
char ** fb_get_field_restrictions(field_name,op_name,op_instance)

(fabform:get-field-restrictions field-name op-name op-instance)
```

Returns the list of resiricted values associated with a field. Returns `(char *) 0` if the field could not be found or if there are no restrictions on the field. Each entry in the list is a pointer to a possible value for the field. The last entry in the list is NULL.

The restriction list returned in the current implementation is in fact the list used internally by *Fabform*, thus any changes to that list will affect the validation of entries for the field. In addition, different fields may share a restriction list (eg. via oneof-class declarations), thus changing a list may affect more than one field.

The Lisp implementation returns a list of strings corresponding to the restrictions. There is no sharing of data in the Lisp implementation, thus the returned list and its contents may be mutated without consequences.

```
int fb_set_field_restrictions(field_name,op_name,op_instance,
                              restrictions,free_storage)
        char *field_name,*op_name,**restrictions;
        int op_instance,free_storage;

(fabform:set-field-restrictions field-name op-name op-instance
                                restrictions)
```

All parameters (and the return value) have the same properties as their counterparts in `fb_set_field_value()`. The restriction list specified by `restrictions` should be a list of pointers to C-style strings of possible values for the field and should be terminated by NULL. `free_storage` should be set if the restriction list and all of its entries were allocated via malloc() and should be free()d when the restriction list is no longer needed by this field (ie. when a new restriction list is specified or when this instantiation of *Fabform* exits).

The Lisp implementation takes a list of strings as the `restrictions`. The entries in the list are copied into newly allocated storage before being installed as the restriction list for the field and will be freed when they are no longer needed.

```
int fb_set_field_comment(field_name,op_name,op_instance,comment)
        char *field_name,*op_name,*comment;
        int op_instance;

(fabform:set-field-comment field-name op-name op-instance
                           comment)
```

Sets the comment printed when the cursor is on a given field. The return value will be 0 if successful, -1 if the specified operation instance could not be found, and -2 if the field could not be found.

```
int fb_place_cursor_on_field(field_name,op_name,op_instance,
                             refresh)
        char *field_name,*op_name;
        int op_instance,refresh;

(fabform:place-cursor-on-field field-name op-name
                               op-instance refresh)
```

Moves the cursor to the specified field. If refresh is non-zero (or non-nil), the screen is updated. Returns 0 if successful, -1 if the operation instance could not be found, -2 if the field could not be found.

```
int fb_delete_region(start_op_name,start_op_instance,end_op_name,
                     end_op_instance,refresh)
        char *start_op_name,*end_op_name;
        int start_op_instance,end_op_instance,refresh;

(fabform:delete-region start-op-name start-op-instance end-op-name
                       end-op-instance refresh)
```

Delete the region of the form beginning with `start_op` until the end of `end_op`. If the cursor is on a field that is deleted by the operation, it will be moved to a new field if possible. The region must begin at the start of a template file and the portion of the form following the end of the region must also begin at the start of a template file. If refresh is non-zero (or non-nil), the screen is updated. Returns 0 if successful, -1 if the start operation instance could not be found, -2 if the end operation instance could not be found, and -3 if a bad region is specified.

```
int fb_insert_region_before(op_name,op_instance,param_file,
                     template_dir,refresh)
        char *op_name,*param_file,*template_dir;
        int start_op_instance,refresh;

(fabform:insert-region-before op-name op-instance param-file
                              template-dir refresh)
```

Inserts the region defined by `param_file` into the form just before the specified operation instance. The operation instance must begin at the start of a template file. `template_dir` specifies where to look for template files referenced in `param_file`. Use of globals or oneof-class defnitions in the specified parameter file will not affect fields already existing in the file, thus their use is discouraged. If refresh is non-zero (or non-nil), the screen is updated. Returns 0 if successful, -1 if the operation instance could not be found, -3 if a bad region is specified, and -4 if the parameter file could not be opened. Very bad things can happen if an error occurs while reading the parameter file - usually resulting in the program exitting with an error message.

```
int fb_insert_region_after(op_name,op_instance,param_file,
                     template_dir,refresh)
        char *op_name,*param_file,*template_dir;
        int start_op_instance,refresh;
```

```
(fabform:insert-region-after op-name op-instance param-file
                             template-dir refresh)
```

Just like fb_insert_region_before except that the region is inserted just after the speficied operation instance. A bad region error will result if the operation following the specified operation instance does not begin at the start of a template file.

```
int fb_set_ctrlx_keymap(key,function,start_action)
        char key;
struct fabform_retstat (*function)();
        int start_action;

(fabform:set-ctrlx-keymap key function-handle start-action)
```

Bind the keystroke control-X key to the specified function. `key` may be any character in the 128 character standard ASCII set. `function` should be a pointer to a function of the form normally attached to a field (i.e. the same interface for receiving and returning values). `start_action` is identical to the field that would have been specified as a part of the function definition list passed to *Fabform* upon startup.

In the Lisp interface, `function-handle` should be an object returned by `fabform:make-fabform-function` or `fabform:defun-fabform-function`. Similarly, `start-action` should be a (possibly null) list of keywords specifying which actions to take before executing the function.

A key binding may be deleted by either defining a new binding or specifying a NULL for `function` (or nil for `function-handle`). The application-defined key bindings take precedence over any bindings that *Fabform* uses, thus applications should be careful about how they use bindings. Also, upper and lower case characters are treated differently.

The key bindings are unique to each invocation of *Fabform*. The keymap is reset to it's default state (i.e. no application-defined bindings) when *Fabform* is invoked. Currently the only method of binding keys immediately after *Fabform* is invoked is to specify a function named "fabform_init" in the function definition list given to *Fabform* upon startup. This init function should then set up the keymaps as desired.

```
int fb_set_esc_keymap(key,function,start_action)
        char key;
```

```
struct fabform_retstat (*function)();
        int start_action;
```

```
(fabform:set-esc-keymap key function-handle start-action)
```

Just like `fb_set_ctrlx_keymap` (`fabform:set-ctrlx-keymap`) but binds keystrokes of the form ESC key instead.

```
int fb_get_movement_repeat_count()
```

```
(fabform:get-movement-repeat-count)
```

Returns the value of the movement repeat count entered using control-U. Useful for functions that are bound to keystrokes which move the cursor via `fb_place_cursor_on_field`, for example.

```
struct fabform_retstat fb_exec_fabform_function(funcdef,field_value,field_name,
                                                op_name,op_inst,filenam)
     struct fabform_functions funcdef;
     char *field_value,*field_name,*op_name,*filenam;
     int op_inst;
```

Executes a function of the type that is attached to a field and returns the correct value even if the function is coded in Lisp (The action field should have the FABFORM_LISP_CALL bit set if a Lisp function is to be called).

There is currently no Lisp implementation of `exec_fabform_function`.

D.6 Linking the *Fabform* Subroutine

The structures and values necessary for defining functions are in the include file "/usr/cafe/include/fabform.h". To create the executable file for an application using *Fabform* as a subroutine, simply specify "/usr/cafe/lib/fabform.a" in the list of files to link in. All of the necessary library functions have already been linked into the fabform.a file.

The Lisp implementation lives in a package called "fabform" in "/usr/cafe/lib/fabform.fasl" or "/usr/cafe/lib/fabform.lisp". Lisp applications written under Franz Extended Common Lisp should probably have something like

```
(require 'fabform "/usr/cafe/lib/fabform.fasl")
```

at the start of the application. Remember to specify the .fasl or .lisp suffix or the wrong file will be loaded.

Support has also been added for applications using Kyoto Common Lisp. These applications should load the file "/usr/cafe/lib/fabform.lsp". Applications that may run under either Franz or KCL should load the .lsp file. Do not, under any circumstances, attempt to directly load "/usr/cafe/lib/fabform.o" under KCL. In addition, the KCL implementation requires the presence of "kcl-ffsupport.o" in /usr/cafe/lib. If you are not running in the cafe environment, you will need to obtain this file and install it in an appropriate location.

There are a few precautions that should be taken by applications that link with the *Fabform* routines. Any global symbols of function names beginning with "fb_" or "screenio_" are reserved for use by *Fabform*. If the application attempts to use any globals in this namespace, disasters are likely to happen. In addition, *Fabform* uses parts of the *curses* and *termcap* libraries, thus applications should be wary of any routines or globals used by those libraries.

Bibliography

[Caplinger85] Caplinger, Michael, "Structured Editor Support for Modularity and Data Abstraction", *ACM SIGPLAN Notices*, Volume 20, Number 7 (July 1985), pages 140-147.

[Fischer84] Fischer, C.N., Pal, Anil, and Stock, Daniel, "The POE Language-Based Editor Project", *ACM SIGPLAN Notices*, Volume 19, Number 5 (May 1984), pages 21-29.

[Garlan84] Garlan, David B. and Miller, Phillip L., "GNOME: An Introductory Programming Environment Based on a Family of Structure Editors", *ACM SIGPLAN Notices*, Volume 19, Number 5 (May 1984), pages 65-72.

[Horgan84] Horgan, J.R. and Moore, D.J., "Techniques for Improving Language-Based Editors", *ACM SIGPLAN Notices*, Volume 19, Number 5 (May 1984), pages 7-14.

[Horwitz85] Horwitz, Susan and Teitelbaum, Tim, "Relations and Attributes: A Symbiotic Basis for Editing Environments", *ACM SIGPLAN Notices*, Volume 20, Number 7 (July 1985), pages 93-106;

[Lederman81] Lederman, Abraham, "A Pascal Structure-Oriented Editor: Design and Implementation Issues", M.I.T. Master's Thesis, 1981.

[Reiss84] Reiss, Steven P., "Graphical Program Development with PECAN Program Development Systems", *ACM SIGPLAN Notices*, Volume 19, Number

5 (May 1984), pages 30-41.

[Rubin83] Rubin, Lisa F., "Syntax-Directed Pretty Printing – A First Step Towards a Syntax-Directed Editor", *IEEE Transactions on Software Engineering*, Volume SE-9, Number 2 (March 1983), pages 119-127.

[Sedayao88] Sedayao, Jeff, "SEPS: A Structured Editor for Process Specification", presented at the CIM-IC Workshop at Stanford University, August 4-5, 1988.

[Teitelbaum81] Teitelbaum, Tim and Reps, Thomas, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communications of the ACM*, Volume 24, Number 9 (September 1981), pages 563-573.

[Teitelbaum84] Teitelbaum, Tim and Reps, Thomas, "The Synthesizer Generator", *ACM SIGPLAN Notices*, Volume 19, Number 5 (May 1984), pages 42-48.

[Zelkowitz84] Zelkowitz, Marvin V., "A Small Contribution to Editing with a Syntax Directed Editor", *ACM SIGPLAN Notices*, Volume 19, Number 5 (May 1984), pages 1-6.